前端撷英馆

Vue.js 权威指南



张耀春 黄轶 王静 苏伟 王槿 殷献勇 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

Vue.js 是一个用来开发 Web 界面的前端库。本书致力于普及国内 Vue.js 技术体系，让更多喜欢前端的人员了解和学习 Vue.js。如果你对 Vue.js 基础知识感兴趣，如果你对源码解析感兴趣，如果你对 Vue.js 2.0 感兴趣，如果你对主流打包工具感兴趣，如果你对如何实践感兴趣，本书都是一本不容错过的以示例代码为引导、知识涵盖全面的最佳选择。

全书一共 30 章，由浅入深地讲解了 Vue.js 基本语法及源码解析。主要内容包括数据绑定、指令、表单控件绑定、过滤器、组件、表单验证、服务通信、路由和视图、vue-cli、测试开发和调试、源码解析及主流打包构建工具等。该书内容全面，讲解细致，示例丰富，适用于各层次的开发者。

图书在版编目（CIP）数据

Vue.js 权威指南 / 张耀春等著. —北京：电子工业出版社，2016.9
（前端撷英馆）
ISBN 978-7-121-28722-0

Ⅰ. ①V… Ⅱ. ①张… Ⅲ. ①网页制作工具－程序设计 Ⅳ. ①TP392.092.2

中国版本图书馆 CIP 数据核字（2016）第 204355 号

策划编辑：张春雨
责任编辑：葛 娜
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　印张：31.75　字数：687 千字
版　　次：2016 年 9 月第 1 版
印　　次：2016 年 9 月第 1 次印刷
定　　价：99.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

推荐序

三年前刚开始写 Vue.js 的第一个原型的时候，我并没有奢望太多。而今天在我打下这行字的时候，Vue.js 在 GitHub 上已经有超过两万五千颗 star，在 npm 上有超过一百万次的下载，并在全球各地拥有十几万的用户。同时，得益于社区的大力支持，我有幸全职投入 Vue.js 的开发，以开源作为我的全职工作。

在设计思想上，Vue.js 尤为注重上手的学习曲线，这也是 Vue.js 吸引很多用户的一个重要原因。但随着近年来产品对前端的需求不断提高，前端工程的复杂度也在不断提升。因此，初学者免不了在实际的生产应用中遇到各种文档中难以覆盖的细节问题。

在国内，Vue.js 被大量行业领先的科技公司在生产中广泛使用，这其中就包括本书作者小春所就职的滴滴出行。本书包含了大量在实践中总结出的 Vue.js 使用经验，并且能看出作者对 Vue.js 的内部实现做了深入的研究，相信能为国内学习 Vue.js 的开发者们提供有价值的帮助。Vue.js 的迅速成长，在很大程度上得益于社区用户与他人分享经验心得的热心。在这里，我也对小春和滴滴公共前端团队为本书付出的心血表示衷心的感谢。

尤雨溪

Vue.js 作者

前　　言

本书是一本全方位讲解 Vue.js，从入门到精通的权威指南。

从本书中你将学到：

- Vue.js 基本语法
- Vue.js 源码解析
- 如何开发一个完整的组件
- 如何集成第三方组件
- 如果构建和调试一个 Vue 的项目
- 主流打包构建工具的使用
- Vue.js 2.0
- Scrat 与 Vue.js 结合

本书读者对象

本书写给从未使用 Vue.js 开发项目或想深入了解 Vue.js 原理的读者，同时也适合热衷于追求新技术、探索新工具的读者。特此声明：本书基础语法讲解基于 Vue.js 1.0 版本，其中涵盖了与其他版本的比较。我们假设读者已经掌握了 HTML 和 CSS，并且熟悉 JavaScript 基础知识。

如何阅读本书

如果你从事 Web 开发工作，之前没有接触过 Vue.js，建议从第 1 章开始仔细阅读，并亲手实践每个章节提供的示例，可以加深理解；如果你已经使用 Vue.js 开发项目，则可以跳过前面基础知识，直接进入源码解析篇，让我们共同探索 Vue.js 是如何实现的，以及有哪些值得借鉴学习的知识；如果你想看看 Vue.js 2.0 都发生了什么转变，请直接进入 Vue.js 2.0 章节阅读；如

果你想了解打包构建工具如何使用，请直接进入工具篇，那里有三款打包工具供选择。希望你阅读愉快。

本书结构

每个章节的开头都会介绍一个概念，帮你了解该章节所讲内容是什么，以便快速了解或准确地找到所关注的内容

在基础知识讲解中，每一节中都会有大量丰富、详尽的示例，方便你更全面地掌握所讲解的知识。

在章节最后还会附加一些常见问题，帮助你快速解决问题并定位问题所在。

目　　录

第 1 章

遇见 Vue.js

滴滴公共前端团队从 2013 年开始接触 React 和 AngularJS，以及后来的 Polymer，在项目实战中踩过了各种坑，参与了一些公司级的组件库开发和复杂业务模块的设计，也在与之配套的工程化闭环里做了很多解决方案。

回过头来看看，这几年的前端开发已经不再是去适配低版本的 PC 浏览器，对于大部分国内一线的移动互联网公司的前端开发者，移动端的前端项目需求尤其强烈，用户体验也一再被大家提及，页面已经不能简单地通过重新渲染来更新数据的频繁变化，后端的一些 MVC 模式也在往前端框架迁移。

在正式学习 Vue.js 之前，我们先和大家简单地回顾一下 MVX。

1.1 MVX 模式是什么

MVC 框架最早出现在 Jaca 领域，然后慢慢在前端开发中也被提到，后来又出现了 MVP，以及现在最成熟的 MVVM，下面我们来简单介绍一下各种模式。

1.1.1 MVC

MVC 是应用最广泛的软件架构之一，一般 MVC 分为：Model（模型）、Controller（控制器）和 View（视图）。这主要是基于分层的目的，让彼此的职责分开，如图 1-1 所示。

View 一般都是通过 Controller 来和 Model 进行联系的。Controller 是 Model 和 View 的协调者，View 和 Model 不直接联系。基本联系都是单向的。

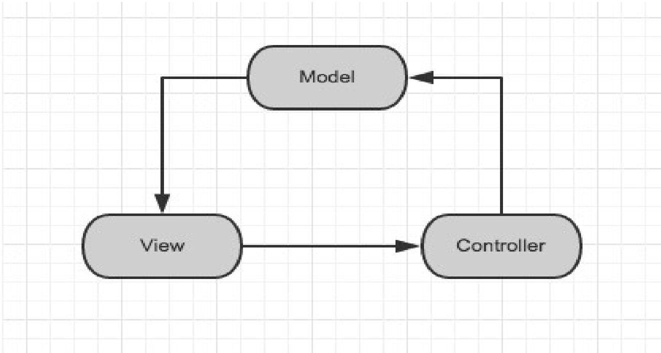


图1-1　MVC通信方式一

那么，用户操作应该放在什么位置，MVC 之间又会有什么变化，如图 1-2 所示。

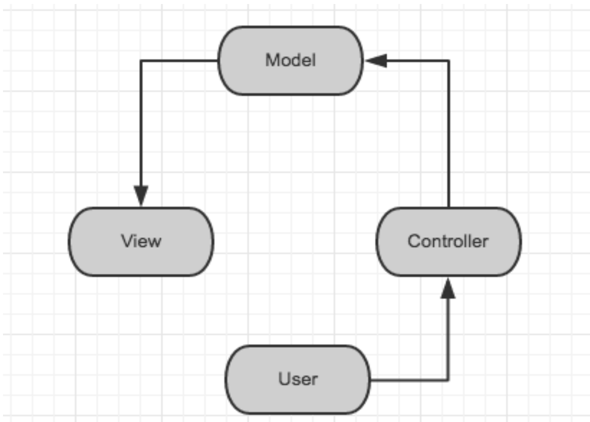


图1-2　MVC通信方式二

用户（User）通过 Controller 来操作 Model 以达到 View 的变化。

1.1.2　MVP

MVP 是从经典的 MVC 模式演变而来的，它们的基本思想有相通的地方：Controller/Presenter 负责逻辑的处理，Model 提供数据，View 负责显示。

在 MVP 中，Presenter 完全把 View 和 Model 进行了分离，主要的程序逻辑在 Presenter 里实现。而且，Presenter 与具体的 View 是没有直接关联的，而是通过定义好的接口进行交互，从而使得在变更 View 的时候可以保持 Presenter 不变。MVP 通信方式如图 1-3 所示。

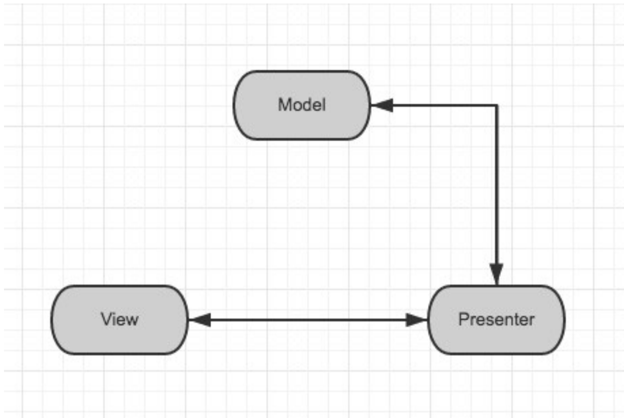


图1-3　MVP通信方式

1.1.3　MVVM

MVVM 代表框架有：知名度相对偏低的 Knockout、早期的 Ember.js、目前比较火热的来自 Google 的 AngularJS，以及我们今天要讲的 Vue.js。

相比前面两种模式，MVVM 只是把 MVC 的 Controller 和 MVP 的 Presenter 改成了 ViewModel。View 的变化会自动更新到 ViewModel，ViewModel 的变化也会自动同步到 View 上显示。

这种自动同步是因为 ViewModel 中的属性实现了 Observer，当属性变更时都能触发对应的操作，如图 1-4 所示。

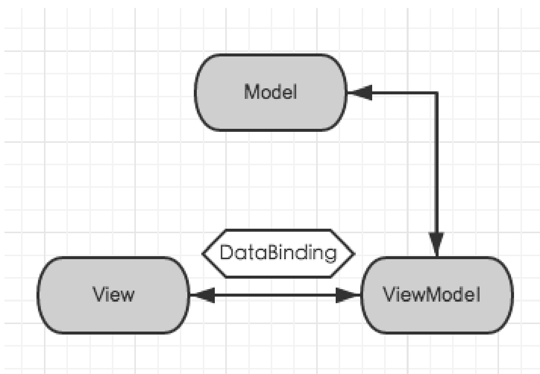


图1-4　用户操作影响

1.2　Vue.js 是什么

Vue.js 不是一个框架——它只聚焦视图层，是一个构建数据驱动的 Web 界面的库。Vue.js 通过简单的 API 提供高效的数据绑定和灵活的组件系统。

先看看 Vue.js 的特性。

1. 确实轻量

除了以 MVP 模式代表的 Riot.js 外，Vue.js 已经算是前端库里面体积非常小的，且不依赖其他基础库。

2. 数据绑定

对于一些富交互、状态机类似的前端 UI 界面，数据绑定非常简单、方便。

3. 指令

类似于 AngularJS，可以用一些内置的简单指令（v-*），也可以自定义指令，通过对应表达式值的变化就可以修改对应的 DOM。

4. 插件化

Vue.js 核心包不包含 Router、AJAX、表单验证等功能，但是可以非常方便地加载对应的插件，后续章节我们会做完整的补充说明。

1.2.1　Vue.js 与其他框架的区别

相信很多读者都有一些其他框架（比如 AngularJS）的学习或者应用背景，本节将以对比方式来介绍各自的特点。

1. 与 AngularJS 的区别

首先要提到的肯定是 AngularJS，它来自 Google，是目前国内最火的前端框架之一，应用于 PC 类的复杂交互系统，我们内部也产出了一套基于它的 PC UI 组件库。那两者到底有什么区别呢？

相同点：

- 都支持指令——内置指令和自定义指令。
- 都支持过滤器——内置过滤器和自定义过滤器。

- 都支持双向绑定。
- 都不支持低端浏览器（比如 IE6/7/8）：
  - Vue.js 使用比如 Array.isArray 的 ES 5 特性。
  - AngularJS 1.3 开始不支持 IE 8。

不同点：

- AngularJS 的学习成本比较高，比如增加了 Dependency Injection 特性，而 Vue.js 本身提供的 API 都比较简单、直观。
- 在性能上，AngularJS 依赖对数据做脏检查，所以 Watcher 越多越慢。Vue.js 使用基于依赖追踪的观察并且使用异步队列更新，所有的数据都是独立触发的。对于庞大的应用来说，这个优化差异还是比较明显的。

2. 与 React 的区别

第二个要提到的便是 React，来自 Facebook，在国内已经完全复制 AngularJS 的热潮，成为目前受关注度很高的前端框架。为了方便没有用过 React 的同学理解，我们用 React 来编写一个 Footer 组件，代码示例如下：

```
<-- components/FooterView.jsx -->
/** @jsx React.DOM */
var FooterView = React.createClass({
  render: function () {
    return (
      <footer>
        <p>DDFE love Vue.js</p>
      </footer>
    )
  }
});
```

相同点：

- React 采用特殊的 JSX 语法，Vue.js 在组件开发中也推崇编写.vue 特殊文件格式，对文件内容都有一些约定，两者都需要编译后使用。
- 中心思想相同：一切都是组件，组件实例之间可以嵌套。
- 都提供合理的钩子函数，可以让开发者定制化地去处理需求。

- 都不内置类似 AJAX、Router 等功能到核心包，而是以其他方式（插件）加载。
- 在组件开发中都支持 mixins 的特性，具体内容在第 11 章中会进行介绍。

不同点：

- React 依赖 Virtual DOM，而 Vue.js 使用的是 DOM 模板。React 采用的 Virtual DOM 会对渲染出来的结果做脏检查。
- Vue.js 在模板中提供了指令、过滤器等，可以非常方便、快捷地操作 DOM。
- 虽然在第 18 章 Vue.js 2.0 中也会提到支持 Virtual DOM，但是两者还是有差异的。

3. 与 Knockout 的区别

Knockout 也是非常轻量的，甚至兼容 IE 6+的 MVVM 框架。

可能有部分人不熟悉 Knockout，代码示例如下：

```
<input data-bind="value: firstName">
<input data-bind="value:lastName">
<span data-bind="text: fullName"></span>
<script>
var ViewModel = function(first, last) {
  this.firstName = ko.observable(first);
  this.lastName = ko.observable(last);

  this.fullName = ko.computed(function() {
      return this.firstName() + " " + this.lastName();
  }, this);
};

ko.applyBindings(new ViewModel("DDFE", "FE"));
</script>
```

相同点：

- 都用到了数据和 DOM 元素绑定。
- DOM 元素都是基于模板的。
- 都追求 UI 和数据关联，自动刷新。
- 都支持依赖跟踪。

不同点：

- Knockout 的所有可观测属性都需要手动用 observable 方法来初始化，并且需要用函数调用的方式来操作数据。
- Knockout 没有 ViewModel 之间作用域的继承。

4. 与 Ractive.js 的区别

Ractive.js 和 Vue.js API 很像，通过实例化一个 Ractive 类，传一个元素和一些数据、模板等，但是它用字符串模板，数据模型和 Knockout 一样用 get 和 set，本身代码体积较大。

可能大部分人还不熟悉 Ractive.js，代码示例如下：

```
<!--容器占位 -->
<div id="DDFE"></div>
<script>
  var ractive = new Ractive({
    el: "#DDFE",
    template: "<p>我们是一群热爱分享的前端同学，我们来自 {{from}}</p>",
    data: {from: "DDFE"}
  })
</script>
```

5. 与 Polymer 的区别

很多人可能接触 Polymer 不多，它是在 2013 年 Google I/O 之后推出的，提出了 Web Component 早期的很多规范性方案，如 HTML Imports、Shadow DOM、数据绑定等。不过，由于后续的新版本对之前的冲击比较大，也一度受到早期开发者的抱怨。我们先来看一个具体的例子，代码如下：

```
<!--我们之前基于 co-http 的全局封装的 didi-http 模块 -->
<link rel="import" href="../../components/core-ajax/core-ajax.html">
<polymer-element name="didi-http">
  <template>
    <core-ajax id="ajax"
      auto
      url="{{ url }}"
      on-core-response="{{ onResponse }}"
      on-core-error="{{ onError }}"
      handleAs="json"
      withCredentials="true">
    </core-ajax>
  </template>
```

```
  <script>
    (function () {
     Polymer('didi-http', {
         publish: {
            url: {
              value: '',
              reflect: false
            }
         },
         type: {},
         onResponse: function (e, data) {
           this.fire('success', data.response);
         },
         onError: function (data) {
           this.fire('error', data);
         }
     });
    }());
  </script>
</polymer-element>
```

相同点：

- 都支持数据绑定。
- 与 Vue.js 推崇的组件文件都是以.vue 后缀组织结构类似，在 Polymer 中也是把 template、script 都放在一个文件里面。

不同点：

- Polymer 主要推崇 Web Component 标准化，所以会依赖浏览器环境的特性支持，如果不支持就需要加载对应的 Polyfill。
- Polymer 代码体积较大，无法做到轻量级。

6. 与 Backbone.js 的区别

定位不同，Vue.js 专注于 View，而 Backbone 除了 View 之外，还提供了 Collection、Model 及 Router。Vue.js 拥有数据绑定，而 Backbone 需要手动通过事件来操作 DOM。

7. 与 Riot 的区别

作为 React-like 的 MVP 框架的代表，在 Riot 的官方 Git 上有一个框架大小比较列表，Riot

以不到 10KB 的大小稳居第一，超越第二的 Vue.js。我们先来看一个官方的 timer 示例，代码如下：

```
<timer></timer>

<script src="timer.tag" type="riot/tag"></script>
<script src="https://rawgit.com/riot/riot/master/riot%2Bcompiler.min.js"></script>

<script> riot.mount('timer', { start: 0 }) </script>

<!-- timer.tag 内容 -->
<timer>
  <p>DDFE 个数: { time }</p>

  <script>
    this.time = opts.start || 0
    tick() {
      this.update({ time: ++this.time })
    }
    var timer = setInterval(this.tick, 1000)
    this.on('unmount', function() {
      clearInterval(timer)
    })
  </script>

</timer>
```

相同点：

- API 设计简单而专注，学习成本低。
- 提供自定义的生命周期钩子，方便开发者灵活使用。
- 与主流的工具集成度比较高，支持与各种预编译工具集成。
- 组件化思想，而且将 HTML 和 JS、CSS 混在一个组件中。
- 都只更新变化了的元素。

不同点：

- Riot 内置路由功能、设计支持 Virtual DOM。
- Riot 支持自定义标签，将标签内容放在.tag 文件中，使用 script 特殊的 type="riot/tag"来

加载编译。

- 和 Polymer 的初衷一样，推崇 Web Component 标准化，但是不依赖冗余的 Polyfill。
- Riot 支持服务端渲染，在后续的章节中我们也会介绍 Vue.js 2.0 支持情况。
- Riot 默认单向绑定。

1.2.2 如何使用 Vue.js

上节我们通过对比一些比较熟知的框架，了解了 Vue.js 支持的一些特性和优势，下面简单地来实践一下。

1. 安装

（1）script

如果项目直接通过 script 加载 CDN 文件，代码示例如下：

```
<script src="http://webapp.didistatic.com/static/webapp/shield/z/vue/vue/1.0.24/vue.
min.js"></script>
```

（2）npm

如果项目基于 npm 管理依赖，则可以使用 npm 来安装 Vue，执行如下命令：

```
$ npm i vue -save-dev
```

（3）bower

如果项目基于 bower 管理依赖，则可以使用 bower 来安装 Vue，执行如下命令：

```
$ bower i vue -save-dev
```

2. 第一个 Hello World 程序

每一次学习新框架，都必将经历过 Hello World 程序，我们用 Vue.js 来输出一个微信内滴滴打车的 WebApp 首页 Tab，代码示例如下：

```
<div id="didi-navigator">
  <ul>
    <li v-for="tab in tabs">
      {{ tab.text }}
    </li>
  </ul>
```

```
</div>

new Vue({
  el: '#didi-navigator',
  data: {
    todos: [
      { text: '巴士' },
      { text: '快车' },
      { text: '专车' },
      { text: '顺风车' },
      { text: '出租车' },
      { text: '代驾' }
    ]
  }
})
```

1.2.3　Vue.js 的发展历史

Vue.js 正式发布于 2014 年 2 月，对于目前的 Vue.js：

- 在开发人数上，覆盖 70 多贡献者。
- 在受关注度上，GitHub 拥有 20000 多 Star。

从脚手架、构建、插件化、组件化，到编辑器工具、浏览器插件等，基本涵盖了从开发到测试等多个环节。

Vue.js 的发展里程碑如下：

- 2013 年 12 月 24 日，发布 0.7.0。
- 2014 年 1 月 27 日，发布 0.8.0。
- 2014 年 2 月 25 日，发布 0.9.0。
- 2014 年 3 月 24 日，发布 0.10.0。
- 2015 年 10 月 27 日，正式发布 1.0.0。
- 2016 年 4 月 27 日，发布 2.0 的 preview 版本。

目前推荐使用比较稳定的 1.0.24 版本。

第 2 章 数据绑定

数据绑定是将数据和视图相关联，当数据发生变化时，可以自动更新视图。本章将介绍 Vue.js 中数据绑定的语法。

2.1 语法

2.1.1 插值

文本插值是最基本的形式，使用双大括号{{ }}（类似于 Mustache，所以本文中称作 Mustache 标签），代码示例如下：

```
<span>Text: {{text}}</span>
```

例子中的标签{{text}}将会被相应的数据对象 text 属性的值替换掉，当 text 的值改变时，文本中的值也会联动地发生变化。有时候只需渲染一次数据，后续数据变化不再关心，可以通过“*”实现，代码示例如下：

```
<span>Text: {{*text}} </span>
```

双大括号标签会把里面的值全部当作字符串来处理，如果值是 HTML 片段，则可以使用三个大括号来绑定，代码示例如下：

```
<div>Logo: {{{logo}}}</div>
logo : '<span>DDFE</span>'
```

双大括号标签还可以放在 HTML 标签内，示例如下：

```
<li data-id='{{id}}'></li>
```

总之，Vue.js 提供了一系列文本渲染方式，足够我们应对日常的模板渲染情况。需要注意

的是，Vue 指令和自身特性内是不可以插值的，如果用错了地方，Vue.js 会发出警告。

2.1.2　表达式

Mustache 标签也接受表达式形式的值，表达式可由 JavaScript 表达式和过滤器构成。过滤器可以没有，也可以有多个。

表达式是各种数值、变量、运算符的综合体。简单的表达式可以是常量或者变量名称。表达式的值是其运算结果，代码示例如下：

```
<!--JS 表达式 -->
{{ cents/100 }}   // 在原值的基础上除以 100
{{ true? 1 : 0 }}  // 值为真，则渲染出 1，否则渲染出 0
{{ example.split(",") }}

<!--无效示例-->
{{var logo = 'DDFE'}}  // 这是语句，不是表达式
{{if(true) return 'DDFE'}}  // 条件控制语句是不支持的，可以使用三元式
```

类似于 Linux 中的管道，Vue.js 允许在表达式后面添加过滤符，代码示例如下：

```
{{example | toUpperCase}}
```

这里 toUpperCase 就是过滤器，其本质是一个 JS 函数，返回字符串的全大写形式。Vue.js 允许过滤器串联，代码示例如下：

```
{{example | filterA | filterB}}
```

过滤器还支持传入参数，代码示例如下：

```
{{example | filter a b}}
```

这里 a 和 b 均为参数、用空格隔开。

Vue.js 还提供了许多内置的过滤器，第 6 章将对此进行详细介绍。

2.1.3　指令

指令是带有 v-前缀的特殊特性，其值限定为绑定表达式，也就是 JavaScript 表达式和过滤器。指令的作用是当表达式的值发生变化时，将这个变化也反映到 DOM 上。代码示例如下：

```
<div v-if="show">DDFE</div>
```

当 show 为 true 时，展示 DDFE 字样，否则不展示。还有一些指令的语法稍有不同，在指

令和表达式之间插入一个参数，用冒号分隔，如 v-bind 指令。代码示例如下：

```
<a v-bind:href="url"></a>
<div v-bind:click="action"></div>
```

2.2　分隔符

Vue.js 中数据绑定的语法被设计为可配置的。如果不习惯 Mustache 风格的语法，则可以自己设置。

我们可以在 Vue.config 中配置绑定的语法。Vue.config 是一个对象，包含了 Vue.js 的所有全局配置，可以在 Vue 实例化前修改其中的属性。分隔符在 Vue.config 中源码定义如下：

```
<!--源码目录 src/config.js-->
let delimiters = ['{{', '}}']
let unsafeDelimiters = ['{{{', '}}}']
```

1. delimiters

```
Vue.config.delimiters = ["<%" ,"%>"]
```

如果修改了默认的文本插值的分隔符，则文本插值的语法由{{example}}变为<%example%>。

2. unsafeDelimiters

```
Vue.config.unsafeDelimiters = ["<$", "$>"]
```

如果修改了默认的 HTML 插值的分隔符，则 HTML 插值的语法由{{{example}}}变为<$emample$>。

第 3 章 指令

指令（Directive）是特殊的带有前缀 v-的特性。指令的值限定为绑定表达式，指令的职责就是当其表达式的值改变时把某些特殊的行为应用到 DOM 上。

3.1 内部指令

首先来看看和原生 HTML 标签相似的一组内置指令，这组指令非常容易记忆，因为仅仅是在原生标签前面加上了 v-前缀，如图 3-1 所示。

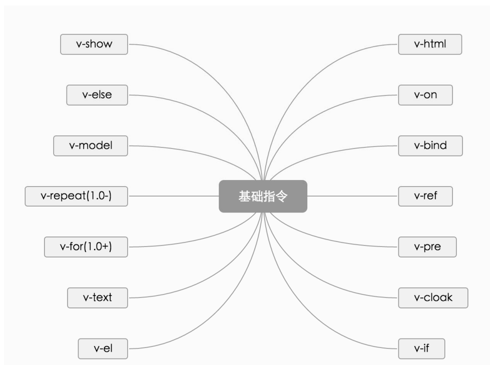


图3-1 内部指令

3.1.1 v-if

v-if指令可以完全根据表达式的值在DOM中生成或移除一个元素。如果 v-if 表达式赋值为 false，那么对应的元素就会从 DOM 中移除；否则，对应元素的一个克隆将被重新插入 DOM 中。代码示例如下：

```
<body class="native">
  <div id="example">
    <p v-if=greeting”>Hello</p>
  </div>
</body>
<script>
var exampleVM2 = new Vue({
  el: '#example',
  data: {
```

```
    greeting: false
  }
})
</script>
```

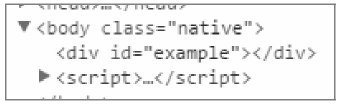


图3-2　v-if

效果如图 3-2 所示。

因为 v-if 是一个指令，需要将它添加到一个元素上。但是如果想切换多个元素，则可以把<template>元素当作包装元素，并在其上使用 v-if，最终的渲染结果不会包含它。代码示例如下：

```
<template v-if="ok">
  <h1>Title</h1>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</template>
```

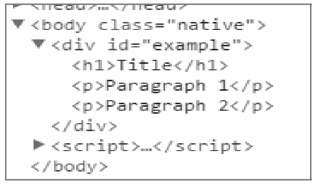


图3-3　v-if

效果如图 3-3 所示。

3.1.2　v–show

v-show 指令是根据表达式的值来显示或者隐藏 HTML 元素。当 v-show 赋值为 false 时，元素将被隐藏。查看 DOM 时，会发现元素上多了一个内联样式 style="display: none"。代码示例如下：

```
<body>
  <input type="text" v-model="message" placeholder="edit me"><body class="native">
  <div id="example">
    <p v-show="greeting">Hello!</p>
  </div>
</body>
<script>
  var exampleVM2 = new Vue({
    el: '#example',
    data: {
      greeting: false
    }
  })
</script>
```

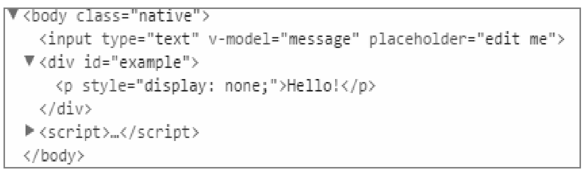


图3-4　v-show

效果如图 3-4 所示。

注：v-show 不支持<template>语法。

在切换 v-if 模块时，Vue.js 有一个局部编译/卸载过程，因为 v-if 中的模板可能包括数据绑定或子组件。v-if 是真实的条件渲染，因为它会确保条件块在切换时合适地销毁与重建条件块内的事件监听器和子组件。

v-if 是惰性的——如果初始渲染时条件为假，则什么也不做，在条件第一次变为真时才开始局部编译（编译会被缓存起来）。

相比之下，v-show 简单得多——元素始终被编译并保留，只是简单地基于 CSS 切换。

一般来说，v-if 有更高的切换消耗，而 v-show 有更高的初始渲染消耗。因此，如果需要频繁地切换，则使用 v-show 较好；如果在运行时条件不大可能改变，则使用 v-if 较好。

3.1.3　v-else

顾名思义，v-else 就是 JavaScript 中 else 的意思，它必须跟着 v-if 或 v-show，充当 else 功能。代码示例如下：

```
<body class="native">
  <div id="example">
    <p v-if="ok">我是对的</p>
    <p v-else="ok">我是错的</p>
  </div>
</body>
<script>
  var exampleVM2 = new Vue({
    el: '#example',
    data: {
      ok: false
    }
  })
</script>
```

将 v-show 用在组件上时，因为指令的优先级 v-else 会出现问题，所以不要这样做。代码示例如下：

```
<custom-component v-show="condition"></custom-component>
<p v-else>这可能也是一个组件</p>
```

我们可以用另一个 v-show 替换 v-else，代码示例如下：

```
<custom-component v-show="condition"></custom-component>
<p v-show="!condition">这可能也是一个组件</p>
```

3.1.4　v–model

v-model 指令用来在 input、select、text、checkbox、radio 等表单控件元素上创建双向数据绑定。根据控件类型 v-model 自动选取正确的方法更新元素。尽管有点神奇，但是 v-model 不过是语法糖，在用户输入事件中更新数据，以及特别处理一些极端例子。代码示例如下：

```
<body id="example">
  <form>
    姓名：
    <input type="text" v-model="data.name" placeholder="">
    </br/>
    性别：
    <input type="radio" id="one" value="One" v-model="data.sex">
    <label for="man">男</label>
    <input type="radio" id="two" value="Two" v-model="data.sex">
    <label for="male">女</label>
    </br/>
    兴趣：
    <input type="checkbox" id="jack" value="book" v-model="data.interest">
    <label for="jack">阅读</label>
    <input type="checkbox" id="john" value="swim" v-model="data.interest">
    <label for="john">游泳</label>
    <input type="checkbox" id="mike" value="game" v-model="data.interest">
    <label for="mike">游戏</label>
    <input type="checkbox" id="mike" value="song" v-model="data.interest">
    <label for="mike">唱歌</label>
    <br>
    身份：
    <select v-model="data.identity">
      <option value="teacher" selected>教师</option>
      <option value="doctor" >医生</option>
      <option value="lawyer" >律师</option>
    </select>
  </form>
</body>
<script>
  new Vue({
```

```
    el: '#example',
    data: {
      data:{
        name:"",
        sex:"",
        interest:[],
        identity:''
      }
    }
  })
</script>
```

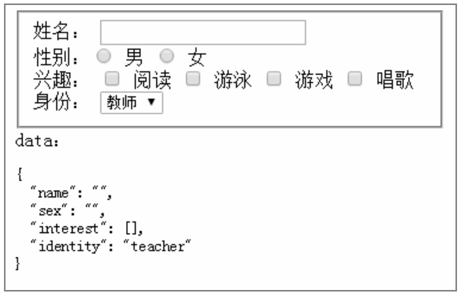


图3-5　v-model

效果如图 3-5 所示。

除了以上用法，在 v-model 指令后面还可以添加多个参数（number、lazy、debounce）。

1. number

如果想将用户的输入自动转换为 Number 类型（如果原值的转换结果为 NaN，则返回原值），则可以添加一个 number 特性。

2. lazy

在默认情况下，v-model 在 input 事件中同步输入框的值与数据，我们可以添加一个 lazy 特性，从而将数据改到在 change 事件中发生。代码示例如下：

```
<body id="example">
  <input v-model="msg" lazy><br/>
  {{msg}}
</body>
<script>
  var exampleVM2 = new Vue({
    el: '#example',
    data: {
      msg: '内容是在 change 事件后才改变的~'
    }
  })
</script>
```

我们在 input 输入框中输入“依然没变”，虽然触发了 input 事件，但是因为加入了 lazy 属性，msg 的值一直没有发生变化。效果如图 3-6 所示。

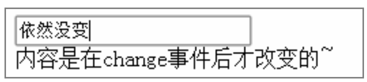


图3-6　lazy

3. debounce

设置一个最小的延时，在每次敲击之后延时同步输入框的值与数据。如果每次更新都要进行高耗操作（例如，在 input 中输入内容时要随时发送 AJAX 请求），那么它较为有用。代码示例如下：

```
<body id="example">
  <input v-model="msg" debounce="5000"><br/>
  {{msg}}
</body>
<script>
  var exampleVM2 = new Vue({
    el: '#example',
    data: {
      msg: '内容是在 5000ms 后才改变的~'
    }
  })
</script>
```

在 5000ms 内我们将输入框的内容清空，msg 的值没有马上改变，还依然保持着“内容是在 5000ms 后才改变的~”，效果如图 3-7 所示。

5000ms 后内容才被清空，效果如图 3-8 所示。

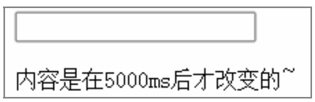


图3-7　5000ms内效果

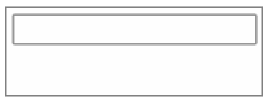


图3-8　5000ms后效果

3.1.5　v-repeat

我们可以使用 v-repeat 指令基于源数据重复渲染元素。对于数组中的每一个对象，该指令都会创建一个子 Vue 实例。这些子实例继承父类的所有数据，所以在重复的元素中，我们可以访问重复实例和父实例的属性。此外，我们也可以使用$index 来呈现相对应的数组索引。代码示例如下：

```
<body id="example">
  <ul id="demo">
    <li v-repeat="items" class="item-{{$index}}">
      {{$index}} - {{parentMsg}} {{childMsg}}
```

```
    </li>
  </ul>
</body>
<script>
  var demo = new Vue({
    el: '#demo',
    data: {
      parentMsg: '滴滴',
      items: [
        { childMsg: '顺风车' },
        { childMsg: '专车' }
      ]
    }
  })
</script>
```

效果如图 3-9 所示。

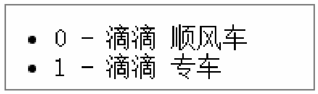


图3-9　v-repeat

有时我们可能想重复一个包含多个 DOM 元素的块，在这种情况下，则可以使用<template>标签来包装重复片段。这里的<template>标签只充当一个语义包装器。代码示例如下：

```
<ul>
  <template v-repeat="list">
    <li>{{msg}}</li>
    <li class="divider"></li>
  </template>
</ul>
```

对于含有原始值的数组，我们可以简单地通过$value 来获取值。代码示例如下：

```
<body id="example">
  <ul id="tags">
    <li v-repeat="tags">
      {{$value}}
    </li>
  </ul>
</body>
<script>
  var demo = new Vue({
    el: '#tags',
    data: {
```

```
    tags: ['JavaScript', 'MVVM', 'Vue.js']
  }
})
</script>
```

效果如图 3-10 所示。



图3-10 $value

有时我们更希望显式地访问变量，而不是隐式地回退到父作用域，这时可以通过使用别名的方式实现：

```
v-repeat="item in items"
```

注：Vue.js 0.12.8 及以后版本支持 in 分隔符。

当数组数据出现变动时如何检测呢？Vue.js 包装了被观察数组的变异方法，它们能触发视图更新。被包装的方法有：

- push()
- pop()
- shift()
- unshift()
- splice()
- sort()
- reverse()

源码定义如下：

```
<!--源码目录：vue/src/observer/array.js 10 行-->
;[
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]
.forEach(function (method) {
```

```
  // cache original method
  var original = arrayProto[method]
  def(arrayMethods, method, function mutator () {
    // avoid leaking arguments:
    // http://jsperf.com/closure-with-arguments
    var i = arguments.length
    var args = new Array(i)
    while (i--) {
      args[i] = arguments[i]
    }
    var result = original.apply(this, args)
    var ob = this.__ob__
    var inserted
    switch (method) {
      case 'push':
        inserted = args
        break
      case 'unshift':
        inserted = args
        break
      case 'splice':
        inserted = args.slice(2)
        break
    }
    if (inserted) ob.observeArray(inserted)
    // notify change
    ob.dep.notify()
    return result
  })
})
```

如源码所示，Vue.js 重写这些方法之后，触发了一次 notify。

Vue.js 还增加了两个方法来观测变化：$set、$remove。源码定义如下：

```
<!--源码目录：vue\src\observer\array.js 60 行-->
def(
  arrayProto,
  '$set',
  function $set (index, val) {
    if (index >= this.length) {
      this.length = Number(index) + 1
```

```
    }
    return this.splice(index, 1, val)[0]
  }
)
def(
  arrayProto,
  '$remove',
  function $remove (item) {
    /* istanbul ignore if */
    if (!this.length) return
    var index = indexOf(this, item)
    if (index > -1) {
      return this.splice(index, 1)
    }
  }
)
```

我们应该尽量避免直接设置数据绑定的数组元素，因为这些变化不会被 Vue.js 检测到，因而也不会更新视图渲染。这时，我们可以使用$set 方法：

```
// same as `demo.items[0] = ...` but triggers view update
demo.items.$set(0, { childMsg: 'Changed!'})
```

$remove 是$splice 的语法糖，用于从目标数组中查找并删除元素。因此，不必这样：

```
<span v-text="msg"></span><br/>
<!-- same as -->
<span>{{msg}}</span>
```

只用这样：

```
// remove the item at index 0
demo.items.$remove(0)
```

另外，也可以使用 filter、concat、slice 方法，返回的数组将是一个不同的实例。我们可以用新的数组替换原来的数组。

```
demo.items = demo.items.filter(function (item) {
  return item.childMsg.match(/Hello/)
})
```

在某些情况下，我们有时可能需要用全新对象（例如，通过 API 调用创建的对象）来替换数组。因为在默认情况下，v-repeat 通过数据对象的特征来决定对已有作用域和 DOM 元素的复用程度，这可能导致重新渲染整个列表。但是，如果每个对象都有一个唯一的 ID 属性，便可以使用 track-by 特性给 Vue.js 一个提示，因而 Vue.js 能尽可能地复用已有实例。假定数据为：

```
{
  items: [
    { _uid: '88f869d', ... },
    { _uid: '7496c10', ... }
  ]
}
```

可以这样给出提示，代码示例如下：

```
<div v-repeat ="item in items" track-by="_uid">
  <!-- content -->
</div>
```

在替换数组 items 时，如果 Vue.js 遇到一个包含有_uid: '88f869d'的新对象，那么它知道可以复用这个已有对象的作用域与 DOM 元素。

如果没有唯一的键供追踪，则可以使用 track-by="$index"，它强制让 v-for 进入原位更新模式：片段不会被移动，而是简单地以对应索引的新值刷新。这种模式也能处理数据数组中重复的值。

这让数据替换非常高效，但是也会付出一定的代价。因为这时 DOM 节点不再映射数组元素顺序的改变，不能同步临时状态（比如<input>元素的值），以及组件的私有状态。因此，如果 v-repeat 块包含<input>元素或子组件，则要小心使用 track-by="$index"。

因为 JavaScript 的限制，Vue.js 不能检测到下面数组的变化：

- 直接用索引设置元素，如 vm.items[0] = {}。
- 修改数据的长度，如 vm.items.length = 0。

为了解决前一个问题，Vue.js 扩展了观察数组，我们可以使用上面讲过的$set 方法：

```
// 与 `example1.items[0] = ...` 相同，但是能触发视图更新
vm.items.$set(0, { childMsg: 'Changed!'})
```

至于后一个问题，只需用一个空数组替换 items 即可。

我们也可以使用 v-repeat 遍历一个对象，每一个重复的实例都将有一个特殊的属性$key，对于原始值，我们可以使用$value 来获取。代码示例如下：

```
<body id="example">
  <ul id="repeat-object">
    <li v-repeat="primitiveValues">{{$key}} : {{$value}}</li>
    <li>===</li>
```

```
    <li v-repeat="objectValues">{{$key}} : {{msg}}</li>
  </ul>
</body>
<script>
  var demo = new Vue({
    el: '#repeat-object',
    data: {
      primitiveValues: {
        FirstName: 'DIDI',
        LastName: 'FE',
        Age: 4
      },
      objectValues: {
        one: {
          msg: 'Hello'
        },
        two: {
          msg: 'DIDI FE'
        }
      }
    }
  })
</script>
```

效果如图 3-11 所示。

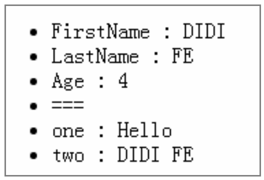


图3-11　遍历对象

注：ECMAScript 5 无法检测到新属性添加到一个对象上或者在对象中删除。要处理这种情况，Vue.js 增加了三种方法：$add(key, value)、$set(key, value)和$delete(key)，这些方法可以用来添加和删除属性，同时触发视图更新。

v-repeat 也支持整数。代码示例如下：

```
<div id="range">
  <div v-repeat="val">Hi! {{$index}}</div>
</div>
```

将模板重复整数次，效果如图 3-12 所示。

也可以将元素重复整数次，代码示例如下：

```
<span v-repeat="n in 10">{{n}}</span>
```

效果如图 3-13 所示。

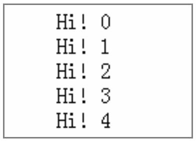


图3-12　重复模板



图3-13　整数

v-repeat 同时还可以和 Vue.js 提供的内置过滤器或排序数据一起使用。

1. filterBy（0.12 版本）

语法：filterBy searchKey [in dataKey...]

用法：

```
<input v-model="searchText">
<ul>
  <li v-repeat="user in users | filterBy searchText in 'name'">{{user.name}}</li>
</ul>
```

数据如下：

```
users: [
  {
    name:'快车',
    tag:'1'
  },
  {
    name:'出租车',
    tag:'2'
  },
  {
    name:'顺风车',
    tag:'3'
  },
  {
    name:'专车',
    tag:'4'
  },
]
```

在<ul>标签中展示了滴滴业务类型，在输入框中输入“快车，<ul>中数据会根据所输入的“快车”，在 users 的 name 字段中过滤出我们需要的信息，并展示出来。效果如图 3-14 所示。

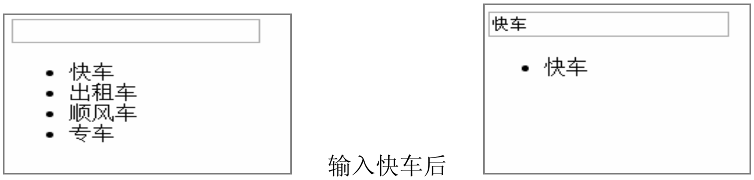


图3-14 filterBy

2. orderBy（0.12 版本）

语法：orderBy sortKey [reverseKey]

用法：

```
<body id="example">
  <ul>
    <li v-repeat="user in users | orderBy field reverse">{{user.name}}</li>
  </ul>
</body>
<script>
  var demo = new Vue({
    el: '#example',
    data: {
      field: 'tag',
      reverse: false,
      users: [
        {
          name:'快车',
          tag: 1
        },
        {
          name:'出租车',
          tag: 3
        },
        {
          name:'顺风车',
          tag: 2
        },
        {
          name:'专车',
          tag: 0
        }
```

```
      ]
    }
  })
</script>
```

在<ul>标签中根据 field 变量代表的 tag 字段正序排列数据，效果如图 3-15 所示。

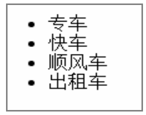


图3-15　orderBy

3.1.6　v–for

我们可以使用 v-for 指令基于源数据重复渲染元素。我们也可以使用$index 来呈现相对应的数组索引，代码示例如下：

```
<body id="example">
<ul id="demo">
<li v-for="item in items" class="item-{{$index}}" class="item-{{$index}}">
{{$index}} -{{parentMessage}} {{item.msg}}}
</li>
</ul>
</body>
<script>
  var demo = new Vue({
    el: '#demo',
parentMessage: '滴滴',
    data: {
      items: [
        { msg: '滴滴顺风车' },
        { msg: '滴滴专车' }
      ]
    }
   })
</script>
```

效果如图 3-16 所示。



图3-16　v-for

v-for 需要特殊的别名，形式为“item in items”（items 是数据数组，item 是当前数组元素的别名）。v-for 在开始时对传入的表达式做了语法分析，不是“item in / of items”的形式，将给出警告信息。Vue.js 1.0.17 及以后版本支持 of 分隔符，更接近 JavaScript 遍历器语法，用法如下：

```
<div v-for="item of items"></div>。
```

源码定义如下：

```
<!--源码目录：vue\src\directive\public\for.js 37行-->
// support "item in/of items" syntax
var inMatch = this.expression.match(/(.*) (?:in|of) (.*)/)
if (inMatch) {
  var itMatch = inMatch[1].match(/\((.*),(.*)\)/)
  if (itMatch) {
    this.iterator = itMatch[1].trim()
    this.alias = itMatch[2].trim()
  } else {
    this.alias = inMatch[1].trim()
  }
  this.expression = inMatch[2]
}

if (!this.alias) {
  process.env.NODE_ENV !== 'production' && warn(
    'Invalid v-for expression "' + this.descriptor.raw + '": ' +
    'alias is required.',
    this.vm
  )
  return
}
```

注：Vue.js 0.12.8 及以后版本支持 in 分隔符。

使用 v-for，将得到一个特殊的作用域，类似于 AngularJS 的隔离作用域，我们需要明确指定 props 属性传递数据，否则在组件内将获取不到数据。对于组件内的<p>标签，我们可以使用<slot>：

```
<my-item  v-for="item in items" :item="item" :index="$index">
<p>{{item.text}}</p>
</my-item>
```

当数组数据出现变动时如何检测呢？Vue.js 包装了被观察数组的变异方法，它们能触发视图更新。被包装的方法有：

- push()
- pop ()
- shift()

- unshift()
- splice()
- sort()
- reverse()

源码定义如下：

```
<!--源码目录：vue/src/observer/array.js 10 行-->
;[
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]
.forEach(function (method) {
  // cache original method
  var original = arrayProto[method]
  def(arrayMethods, method, function mutator () {
    // avoid leaking arguments:
    // http://jsperf.com/closure-with-arguments
    var i = arguments.length
    var args = new Array(i)
    while (i--) {
      args[i] = arguments[i]
    }
    var result = original.apply(this, args)
    var ob = this.__ob__
    var inserted
    switch (method) {
      case 'push':
        inserted = args
        break
      case 'unshift':
        inserted = args
        break
      case 'splice':
```

```
      inserted = args.slice(2)
      break
    }
    if (inserted) ob.observeArray(inserted)
    // notify change
    ob.dep.notify()
    return result
  })
})
```

如源码所示，Vue.js 重写这些方法之后，触发了一次 notify。

Vue.js 还增加了两个方法来观测变化：$set、$remove。源码定义如下：

```
<!--源码目录：vue\src\observer\array.js 60 行-->
def(
  arrayProto,
  '$set',
  function $set (index, val) {
    if (index >= this.length) {
      this.length = Number(index) + 1
    }
    return this.splice(index, 1, val)[0]
  }
)
def(
  arrayProto,
  '$remove',
  function $remove (item) {
    /* istanbul ignore if */
    if (!this.length) return
    var index = indexOf(this, item)
    if (index > -1) {
      return this.splice(index, 1)
    }
  }
)
```

我们应该尽量避免直接设置数据绑定的数组元素，因为这些变化不会被 Vue.js 检测到，因而也不会更新视图渲染。这时，我们可以使用$set 方法：

```
// same as `demo.items[0] = ...` but triggers view update
demo.items.$set(0, { childMsg: 'Changed!'})
```

$remove 是$splice 的语法糖，用于从目标数组中查找并删除元素。因此，不必这样：

```
<span v-text="msg"></span><br/>
<!-- same as -->
<span>{{msg}}</span>
```

只用这样：

```
// remove the item at index 0
demo.items.$remove(0)
```

另外，也可以使用 filter、concat、slice 方法，返回的数组将是一个不同的实例。我们可以用新的数组替换原来的数组。

```
demo.items = demo.items.filter(function (item) {
  return item.childMsg.match(/Hello/)
})
```

在某些情况下，我们有时可能需要用全新对象（例如，通过 API 调用创建的对象）来替换数组。因为在默认情况下，v-for 通过数据对象的特征来决定对已有作用域和 DOM 元素的复用程度，这可能导致重新渲染整个列表。但是，如果每个对象都有一个唯一的 ID 属性，便可以使用 track-by 特性给 Vue.js 一个提示，因而 Vue.js 能尽可能地复用已有实例。假定数据为：

```
{
  items: [
    { _uid: '88f869d', ... },
    { _uid: '7496c10', ... }
  ]
}
```

可以这样给出提示，代码示例如下：

```
<div v-for ="item in items" track-by="_uid">
<!-- content -->
</div>
```

在替换数组 items 时，如果 Vue.js 遇到一个包含有_uid: '88f869d'的新对象，那么它知道可以复用这个已有对象的作用域与 DOM 元素。

如果没有唯一的键供追踪，则可以使用 track-by="$index"，它强制让 v-for 进入原位更新模式：片段不会被移动，而是简单地以对应索引的新值刷新。这种模式也能处理数据数组中重复的值。

这让数据替换非常高效，但是也会付出一定的代价。因为这时 DOM 节点不再映射数组元

素顺序的改变，不能同步临时状态（比如<input>元素的值），以及组件的私有状态。因此，如果 v-for 块包含<input>元素或子组件，则要小心使用 track-by="$index"。

因为 JavaScript 的限制，Vue.js 不能检测到下面数组的变化：

- 直接用索引设置元素，如 vm.items[0] = {}。
- 修改数据的长度，如 vm.items.length = 0。

为了解决前一个问题，Vue.js 扩展了观察数组，我们可以使用上面讲过的$set 方法：

```
// 与 `example1.items[0] = ...` 相同，但是能触发视图更新
vm.items.$set(0, { childMsg: 'Changed!'})
```

至于后一个问题，只需用一个空数组替换 items 即可。

有时我们可能想重复一个包含多个 DOM 元素的块，在这种情况下，则可以使用<template>标签来包装重复片段。这里的<template>标签只充当一个语义包装器。代码示例如下：

```
<ul>
<template v-for="list in lists">
<li>{{list.msg}}</li>
<li class="divider"></li>
</template>
</ul>
```

我们也可以使用 v-for 遍历一个对象，每一个重复的实例都将有一个特殊的属性$key，或者给对象的键值提供一个别名。代码示例如下：

```
<body id="example">
<ul id="repeat-object">
<li v-for ="value in primitiveValues">{{$key}} : {{value}}</li>
<li>===</li>
<li v-for="(key, item )in objectValues">{{key}} : {{item.msg}}</li>
</ul>
</body>
<script>
  var demo = new Vue({
    el: '#repeat-object',
    data: {
     primitiveValues: {
       FirstName: 'DIDI',
       LastName: 'FE',
```

```
        Age: 4
      },
      objectValues: {
        one: {
          msg: 'Hello'
        },
        two: {
          msg: 'DIDI FE'
        }
      }
    }
  })
</script>
```

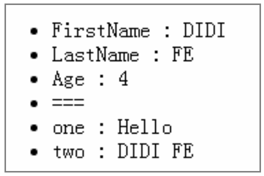


图3-17　遍历对象

效果如图 3-17 所示。

注：ECMAScript 5 无法检测到新属性添加到一个对象上或者在对象中删除。要处理这种情况，Vue.js 增加了三种方法：$add(key,value)、$set(key, value)和$delete(key)，这些方法可以用来添加和删除属性，同时触发视图更新。

v-for 也支持整数。代码示例如下：

```
<div id="range">
<div v-for="n in 10">Hi! {{$index}}</div>
</div>
```

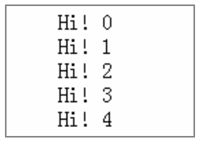


图3-18　重复模板

将模板重复整数次，效果如图 3-18 所示。

v-for 同时还可以和 Vue.js 提供的内置过滤器或排序数据一起使用。

1. filterBy（0.12 版本）

语法：filterBy searchKey [in dataKey...]

用法：

```
<input v-model="searchText">
<ul>
<li v-for="user in users | filterBy searchText in 'name'">{{user.name}}</li>
</ul>
```

数据如下：

```
users: [
  {
```

```
    name:'快车',
    tag:'1'
  },
  {
    name:'出租车',
    tag:'2'
  },
  {
    name:'顺风车',
    tag:'3'
  },
  {
    name:'专车',
    tag:'4'
  },
]
```

在<ul>标签中展示了滴滴业务类型，在输入框中输入“快车，<ul>中数据会根据所输入的“快车”，在 users 的 name 字段中过滤出我们需要的信息，并展示出来。效果如图 3-19 所示。

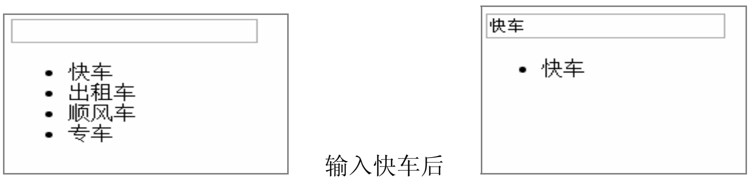


图3-19 filterBy

2. orderBy（0.12 版本）

语法：orderBy sortKey [reverseKey]

用法：

```
<body id="example">
<ul>
<li v-for="user in users | orderBy field reverse">{{user.name}}</li>
</ul>
</body>
<script>
  var demo = new Vue({
    el: '#example',
    data: {
      field: 'tag',
      reverse: false,
      users: [
        {
```

```
          name:'快车',
          tag: 1
        },
        {
          name:'出租车',
          tag: 3
        },
        {
          name:'顺风车',
          tag: 2
        },
        {
          name:'专车',
          tag: 0
        }
      ]
    }
  })
</script>
```

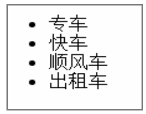


图3-20　orderBy

在<ul>标签中根据 field 变量代表的 tag 字段正序排列数据，效果如图 3-20 所示。

3.1.7　v-text

v-text 指令可以更新元素的 textContent。在内部，{{ Mustache }}插值也被编译为 textNode 的一个 v-text 指令。代码示例如下：

```
<span v-text="msg"></span><br/>
<!-- same as -->
<span>{{msg}}</span>
```

3.1.8　v-html

v-html 指令可以更新元素的 innerHTML。内容按普通 HTML 插入——数据绑定被忽略。如果想复用模板片段，则应当使用 partials。

在内部，{{{ Mustache }}}插值也会被编译为锚节点上的一个 v-html 指令。

注：不建议在网站上直接动态渲染任意 HTML 片段，很容易导致 XSS 攻击。

```
<div v-html="html"></div>
<!-- 相同 -->
<div>{{{html}}}</div>
```

3.1.9 v-bind

v-bind 指令用于响应更新 HTML 特性，将一个或多个 attribute，或者一个组件 prop 动态绑定到表达式。v-bind 可以简写为：

```
<!-- 绑定 attribute -->
<img v-bind:src="imageSrc">
<!-- 缩写 -->
<img :src="imageSrc">
```

在绑定 class 或 style 时，支持其他类型的值，如数组或对象。代码示例如下：

```
<body id="example">
  <div :class="[classA, { classB: isB, classC: isC }]"></div>
</body>
<script>
  var demo = new Vue({
    el: '#example',
    data: {
      classA: 'A',
      isB: false,
      isC: true
    }
  })
</script>
```

效果如图 3-21 所示。

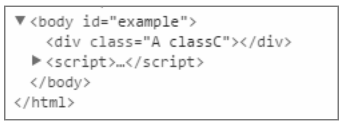


图3-21　v-bind

没有参数时，可以绑定到一个对象。注意，此时 class 和 style 绑定不支持数组和对象（对象 key 会转换为小写）。代码示例如下：

```
<body id="example">
  <div v-bind="{ id: someProp, 'OTHERAttr': otherProp }"></div>
</body>
<script>
  var exampleVM2 = new Vue({
    el: '#example',
    data: {
      someProp: 'idName',
      otherProp: 'prop'
    }
  })
</script>
```

效果如图 3-22 所示。

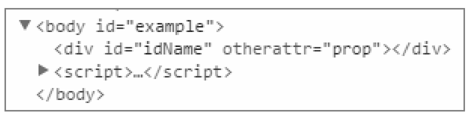


图3-22　bind数组

在绑定 prop 时，prop 必须在子组件中声明。可以用修饰符指定不同的绑定类型。修饰符为：

- .sync ——双向绑定，只能用于 prop 绑定。
- .once ——单次绑定，只能用于 prop 绑定。
- .camel ——将绑定的特性名字转换回驼峰命名。只能用于普通 HTML 特性的绑定，通常用于绑定用驼峰命名的 SVG 特性，比如 viewBox。

```
<!-- prop 绑定，"prop" 必须在 my-component 组件内声明 -->
<my-component :prop="someThing"></my-component>
<!-- 双向 prop 绑定 -->
<my-component :prop.sync="someThing"></my-component>
<!-- 单次 prop 绑定 - -->
<my-component :prop.once="someThing"></my-component>
```

3.1.10　v-on

v-on 指令用于绑定事件监听器。事件类型由参数指定；表达式可以是一个方法的名字或一个内联语句；如果没有修饰符，也可以省略。

使用在普通元素上时，只能监听原生 DOM 事件；使用在自定义元素组件上时，也可以监听子组件触发的自定义事件。

在监听原生 DOM 事件时，如果只定义一个参数，DOM event 为事件的唯一参数；如果在内联语句处理器中访问原生 DOM 事件，则可以用特殊变量$event 把它传入方法。

Vue.js 1.0.11 及以后版本在监听自定义事件时，内联语句可以访问一个$arguments 属性，它是一个数组，包含了传给子组件的$emit 回调的参数。

```
<!-- 方法处理器 -->
<button v-on:click="doThis"></button>
<!-- 内联语句 -->
<button v-on:click="doThat('hello', $event)"></button>
<!-- 缩写 -->
<button @click="doThis"></button>
```

v-on 后面不仅可以跟参数，还可以增加修饰符：

- .stop —— 调用 event.stopPropagation()。

- .prevent —— 调用 event.preventDefault()。
- .capture —— 添加事件侦听器时使用 capture 模式。
- .self —— 只当事件是从侦听器绑定的元素本身触发时才触发回调。
- .{keyCode | keyAlias} —— 只在指定按键上触发回调。Vue.js 提供的键值有：[esc: 27、tab: 9、enter: 13、space: 32、'delete': [8, 46]、up: 38、left: 37、right: 39、down: 40]。

```
<!-- 停止冒泡 -->
<button @click.stop="doThis"></button>
<!-- 阻止默认行为 -->
<button @click.prevent="doThis"></button>
<!-- 阻止默认行为，没有表达式 -->
<form @submit.prevent></form>
<!-- 串联修饰符 -->
<button @click.stop.prevent="doThis">stop</button>
<!-- 键修饰符，键别名 -->
<input @keyup.enter="onEnter">
<!-- 键修饰符，键代码 -->
<input @keyup.13="onEnter">
```

3.1.11 v-ref

在父组件上注册一个子组件的索引，便于直接访问。不需要表达式，必须提供参数 id。可以通过父组件的$refs 对象访问子组件。

当 v-ref 和 v-for 一起使用时，注册的值将是一个数组，包含所有的子组件，对应于绑定数组；如果 v-for 使用在一个对象上，注册的值将是一个对象，包含所有的子组件，对应于绑定对象。

注：因为 HTML 不区分大小写，camelCase 风格的名字比如 v-ref:someRef 将全部转换为小写。可以用 v-ref:some-ref 设置 this.$els.someRef。

3.1.12 v-el

为 DOM 元素注册一个索引，方便通过所属实例的$els 访问这个元素。可以用 v-el:some-el 设置 this.$els.someEl。

```
<span v-el:msg>hello</span>
<span v-el:other-msg>world</span>
```

通过 this.$els 获取相应的 DOM 元素：

```
this.$els.msg.textContent // -> "hello"
this.$els.otherMsg.textContent // -> "world"
```

3.1.13　v-pre

编译时跳过当前元素和它的子元素。可以用来显示原始 Mustache 标签。跳过大量没有指令的节点会加快编译。

3.1.14　v-cloak

v-cloak 这个指令保持在元素上直到关联实例结束编译。AngularJS 也提供了相同的功能。当和 CSS 规则如[v-cloak]{ display: none }一起使用时，这个指令可以隐藏未编译的 Mustache 标签直到实例准备完毕，否则在渲染页面时，有可能用户会先看到 Mustache 标签，然后看到编译后的数据。用法如下：

```
[v-cloak] {
  display: none;
}
<div v-cloak>
  {{ message }}
</div>
```

3.2　自定义指令

使用过 AngularJS 的同学一定知道它的指令是使用 directive(name,factory_function)实现的，格式如下：

```
angular.module('myapp',[],)
.directive(myDeirective, function(){
  return{
    template: '',
    restrict: '',
    template: '',
    replace: ''
    ……
  }
})
```

3.2.1 基础

除了内置指令，Vue.js 也允许注册自定义指令。自定义指令提供一种机制将数据的变化映射为 DOM 行为。

我们来看看 Vue.js 是如何实现的。Vue.js 用 Vue.directive(id,definition)方法注册一个全局自定义指令，它接收两个参数：指令 ID 与定义对象。也可以用组件的 directives 选项注册一个局部自定义指令（此方法相当于 AngularJS restrict 属性值为 A）。

1. 钩子函数

AngularJS 提供了两个函数：compile 和 link，其中编译函数主要负责将作用域和 DOM 进行链接；链接函数用来创建可以操作 DOM 的指令。注意，compile 和 link 选项是互斥的，如果同时设置这两个选项，则会把 compile 返回的函数当作 link 函数，而忽略 link 选项本身。Vue.js 同样也提供了几个钩子函数（都是可选的，相互之间没有制约关系）：

- bind —— 只调用一次，在指令第一次绑定到元素上时调用。
- update —— 在 bind 之后立即以初始值为参数第一次调用，之后每当绑定值变化时调用，参数为新值与旧值。
- unbind —— 只调用一次，在指令从元素上解绑时调用。

```
Vue.directive('my-directive', {
  bind: function () {
    // 准备工作
    // 例如，添加事件处理器或只需要运行一次的高耗任务
  },
  update: function (newValue, oldValue) {
    // 值更新时的工作
    // 也会以初始值为参数调用一次
  },
  unbind: function () {
    // 清理工作
    // 例如，删除 bind()添加的事件监听器
  }
})
```

在注册之后，便可以在 Vue.js 模板中这样用（记着添加前缀 v-）：

```
<div v-my-directive="someValue"></div>
```

当只需要 update 函数时，可以传入一个函数替代定义对象：

```
Vue.directive('my-directive', function (value) {
  // 这个函数用作 update()
})
```

2. 指令实例属性

所有的钩子函数都将被复制到实际的指令对象中，在钩子内 this 指向这个指令对象。这个对象暴露了一些有用的属性：

- el —— 指令绑定的元素。
- vm —— 拥有该指令的上下文 ViewModel。
- expression —— 指令的表达式，不包括参数和过滤器。
- arg —— 指令的参数。
- name —— 指令的名字，不包含前缀。
- modifiers —— 一个对象，包含指令的修饰符。
- descriptor —— 一个对象，包含指令的解析结果。

注：我们应当将这些属性视为只读，不要修改它们。我们也可以给指令对象添加自定义属性，但是注意不要覆盖已有的内部属性。代码示例如下：

```
<body id="example" @click="up" >
  <div id="demo" v-demo:hello.a.b="msg"></div>
</body>
<script>
  Vue.directive('demo', {
    bind: function () {
      console.log('demo bound!')
    },
    update: function (value) {
      this.el.innerHTML =
        'name - '  + this.name + '<br>' +
        'expression - ' + this.expression + '<br>' +
        'argument - '  + this.arg + '<br>' +
        'modifiers - '  + JSON.stringify(this.modifiers) + '<br>' +
        'value - '  + value + '<br>' +
        'vm-msg' + this.vm.msg
```

```
    }
  })
  var demo = new Vue({
    el: '#example',
    data: {
      msg: 'hello!'
    },
    methods: {
      up: function(){
        console.info("click");
      }
    }
  })
</script>
```

效果如图 3-23 所示。

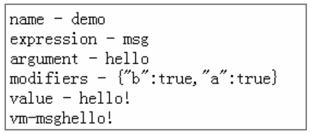


图3-23　指令实例属性

3. 对象字面量

如果指令需要多个值，则可以传入一个 JavaScript 对象字面量。记住，指令可以使用任意合法的 JavaScript 表达式。代码示例如下：

```
<body>
  <div id="demo" v-demo="{ color: 'white', text: 'hello!' }"></div>
</body>
<script>
  Vue.directive('demo', function (value) {
    console.log(value.color) // "white"
    console.log(value.text) // "hello!"
  })
  var demo = new Vue({
    el: '#demo'
  })
</script>
```

效果如图 3-24 所示。



图3-24　对象字面量

4. 字面修饰符

当指令使用了字面修饰符时，它的值将按普通字符串处理并传递给 update 方法。update 方法将只调用一次，因为普通字符串不能响应数据变化。代码示例如下：

```
<body>
  <div id="demo" v-demo.literal="foo bar baz"></div>
</body>
<script>
  Vue.directive('demo', function (value) {
    console.info(value)
  })
  var demo = new Vue({
    el: '#demo'
  })
</script>
```

效果如图 3-25 所示。



图3-25　字面修饰符

5. 元素指令

有时我们想以自定义元素的形式使用指令，而不是以属性的形式。这与 AngularJS 的“E”指令非常相似。元素指令可以看作是一个轻量组件。可以像下面这样注册一个自定义元素指令：

```
<body id="demo">
  <my-directive class="hello" name="hi"></my-directive>
</body>
<script>
  Vue.elementDirective('my-directive', {
    // API 同普通指令
    bind: function () {
      console.info(this.el.className )
      console.info(this.el.getAttribute("name"))
    }
  })
  var demo = new Vue({
    el: '#demo'
  })
</script>
```

元素指令不能接受参数或表达式，但是它可以读取元素的特性，从而决定它的行为。

不同于普通指令，元素指令是终结性的。这意味着，一旦 Vue 遇到一个元素指令，它将跳过该元素及其子元素——只有该元素指令本身可以操作该元素及其子元素。

3.2.2 高级选项

AngularJS 提供了几种方法能够将指令内部的隔离作用域同指令外部的作用域进行数据绑定，如本地作用域属性：@和双向绑定：=以及方法引用：&。

```
scope:{
  ngModel: ‘=’,  // 将 ngModel 同指定对象绑定
  onSend: ‘&’,   // 将引用传递给这个方法
  formName: ‘@’  // 存储与 formName 相关联的字符串
}
```

Vue.js 也允许注册自定义指令。自定义指令提供一种机制将数据的变化映射为 DOM 行为。

1. params

自定义指令可以接受一个 params 数组，指定一个特性列表，Vue 编译器将自动提取绑定元素的这些特性。代码示例如下：

```
<body id="demo">
  <my-directive class="hello" name="hi" a="params"></my-directive>
</body>
<script>
  Vue.elementDirective('my-directive', {
    params: ['a'],
    // API 同普通指令
    bind: function () {
      console.log(this.params.a)
      console.info(this.el.getAttribute("name"))
    }
  })
  var demo = new Vue({
    el: '#demo'
  })
</script>
```

此 API 也支持动态属性。this.params[key]会自动保持更新。另外，可以指定一个回调，在

值变化时调用。代码示例如下：

```
<body id="demo">
  <my-directive class="hello" name="hi" v-bind:a="someValue"></my-directive>
  <input type="text" v-model="someValue"/>
</body>
<script>
  Vue.elementDirective('my-directive', {
    params: ['a'],
    paramWatchers: {
      a: function (val, oldVal) {
        console.log('a changed!')
      }
    }
  })
  var demo = new Vue({
    el: '#demo',
    data:{
      someValue: 'value'
    }
  })
</script>
```

效果如图 3-26 所示。

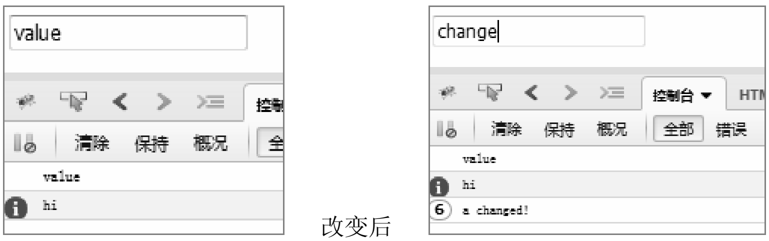


图3-26　params

注：类似于 props，指令参数的名字在 JavaScript 中使用 camelCase 风格，在 HTML 中对应使用 kebab-case 风格。例如，假设在模板中有一个参数 disable-effect，在 JavaScript 中以 disableEffect 访问它。

2. deep

如果自定义指令使用在一个对象上，当对象内部属性变化时要触发 update，则在指令定义对象中指定 deep: true。代码示例如下：

```
<body id="demo">
```

```
  <div v-my-directive="a"></div>
  <button @click="change" >change</button>  {{a.b.c}}
</body>
<script>
  Vue.directive('my-directive', {
    deep: true,
    update: function (obj) {
      // 当 `obj` 的嵌套属性变化时调用
      console.info(obj.b.c)
    }
  })
  var demoVM = new Vue({
    el: '#demo',
    data: {
      a:{ b:{ c:2 } }
    },
    methods:{
      change: function(){
        demoVM.a.b.c = 4;
      }
    }
  })
</script>
```

效果如图 3-27 所示。

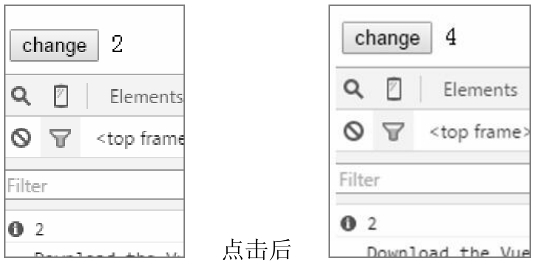


图3-27　deep

3. twoWay

如果指令想向 Vue 实例写回数据，则在指令定义对象中指定 twoWay:true。该选项允许在指令中使用 this.set(value)。代码示例如下：

```
<body id="demo">
  自定义组件：<input v-example="a.b.c" /><br/>
  父作用域：{{ a.b.c}}
</body>
<script>
  Vue.directive('example', {
    twoWay: true,
    bind: function () {
      this.handler = function () {
        // 把数据写回 vm
        // 如果指令这样绑定 v-example="a.b.c"
        // 这里将会给 `vm.a.b.c` 赋值
        this.set(this.el.value)
```

```
    }.bind(this)
    this.el.addEventListener('input', this.handler)
  },
  unbind: function () {
    this.el.removeEventListener('input', this.handler)
  }
})
var demo = new Vue({
  el: '#demo',
  data:{ a:{b:{ c:2 } } }
})
</script>
```

效果如图 3-28 所示。



图3-28　twoWay

4. acceptStatement

传入 acceptStatement:true 可以让自定义指令接受内联语句，就像 v-on 那样。代码示例如下：

```
<body id="demo">
  <div v-my-directive="a++"></div>
  {{a}}
</body>
<script>
  Vue.directive('my-directive', {
      acceptStatement: true,
      update: function (fn) {
        // 传入值是一个函数
        // 在调用它时将在所属实例作用域内计算"a++"语句
        console.info(fn.toString())
        fn()
      }
  })
  var demoVM = new Vue({
      el: '#demo',
      data: {
        a:5
      }
  })
</script>
```

效果如图 3-29 所示。

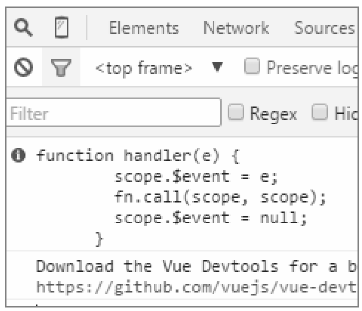


图3-29　acceptStatement

5. Terminal（1.0.19 及以后版本）

Vue 通过递归遍历 DOM 树来编译模块。但是当它遇到 terminal 指令时会停止遍历这个元素

的后代元素，这个指令将接管编译这个元素及其后代元素的任务。v-if 和 v-for 都是 terminal 指令。

编写自定义 terminal 指令是一个高级话题，需要较好地理解 Vue 的编译流程，但并不是说不可能编写自定义 terminal 指令。用 terminal:true 指定自定义 terminal 指令，可能还需要用 Vue.FragmentFactory 来编译 partial。下面是一个自定义 terminal 指令，它编译其内容模板并将结果注入到页面的另一个地方。代码示例如下：

```
<body id="example">
  <div id="modal"></div>
  <div v-inject:modal>
    <h1>header</h1>
    <p>body</p>
    <p>footer</p>
  </div>
  </body>
<script>
  var FragmentFactory = Vue.FragmentFactory
  var remove = Vue.util.remove
  var createAnchor = Vue.util.createAnchor
  Vue.directive('inject', {
    terminal: true,
    bind: function () {
        var container = document.getElementById(this.arg)
        this.anchor = createAnchor('v-inject')
        container.appendChild(this.anchor)
        remove(this.el)
        var factory = new FragmentFactory(this.vm, this.el)
        this.frag = factory.create(this._host, this._scope, this._frag)
        this.frag.before(this.anchor)
    },
    unbind: function () {
        this.frag.remove()
        remove(this.anchor)
    }
  })
  // 创建根实例
  new Vue({
    el: '#example'
  })
</script>
```

效果图如 3-30 所示。

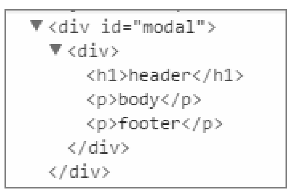


图3-30　terminal

如果想编写自定义 terminal 指令，建议通读内置 terminal 指令的源码，如 v-if 和 v-for，以便更好地了解 Vue.js 的内部机制。

6. priority

可以给指令指定一个优先级。如果没有指定优先级，普通指令默认是 1000，terminal 指令默认是 2000。同一个元素上优先级高的指令会比其他指令处理得早一些，优先级一样的指令按照它在元素特性列表中出现的顺序依次处理，但是不能保证这个顺序在不同的浏览器中是一致的。

另外，流程控制指令 v-if 和 v-for 在编译过程中始终拥有最高的优先级。

3.3　内部指令解析

前面两节分别讲述了内部指令和如何自定义指令，下面我们来简单看一下 Vue.js 的内部指令是如何实现的。源码定义如下：

```
<!--源码目录：src/directive/public/show.js -->
export default {
  bind () {
    // 先存储下一个兄弟节点（如果有），因为 v-else 会和 v-show 绑定使用
    var next = this.el.nextElementSibling
    if (next && getAttr(next, 'v-else') !== null) {
      this.elseEl = next
    }
  },
  update (value) {
    this.apply(this.el, value)
    if (this.elseEl) {
      this.apply(this.elseEl, !value)
    }
  },
  apply (el, value) {
    if (inDoc(el)) {
      applyTransition(el, value ? 1 : -1, toggle, this.vm)
    } else {
      toggle()
    }
    function toggle () {
      el.style.display = value ? '' : 'none'
    }
  }
}
```

再看一下 model 是如何实现的。源码定义如下：

```
<!--源码目录：src/directive/public/model.js -->
export default {
  priority: MODEL,
  twoWay: true,
  handlers: handlers,
  params: ['lazy', 'number', 'debounce'],
  bind () {
    this.checkFilters()
    if (this.hasRead && !this.hasWrite) {
      // 友情提示：此处省略
    }
    var el = this.el
    var tag = el.tagName
    var handler
    if (tag === 'INPUT') {
      handler = handlers[el.type] || handlers.text
    } else if (tag === 'SELECT') {
      handler = handlers.select
    } else if (tag === 'TEXTAREA') {
      handler = handlers.text
    } else {
      process.env.NODE_ENV !== 'production' && warn(
        'v-model does not support element type: ' + tag,
        this.vm
      )
      return
    }
    el.__v_model = this
    handler.bind.call(this)
    this.update = handler.update
    this._unbind = handler.unbind
  },
  checkFilters () {
    var filters = this.filters
    if (!filters) return
    var i = filters.length
    while (i--) {
      var filter = resolveAsset(this.vm.$options, 'filters', filters[i].name)
      if (typeof filter === 'function' || filter.read) {
        this.hasRead = true
      }
      if (filter.write) {
```

```
        this.hasWrite = true
      }
    }
  },
  unbind () {
    this.el.__v_model = null
    this._unbind && this._unbind()
  }
}

<!--源码目录：src/directive/public/radio.js -->
export default {
  bind () {
    var self = this
    var el = this.el
    this.getValue = function () {
      // value overwrite via v-bind:value
      if (el.hasOwnProperty('_value')) {
        return el._value
      }
      var val = el.value
      if (self.params.number) {
        val = toNumber(val)
      }
      return val
    }
    this.listener = function () {
      self.set(self.getValue())
    }
    this.on('change', this.listener)
    if (el.hasAttribute('checked')) {
      this.afterBind = this.listener
    }
  },
  update (value) {
    this.el.checked = looseEqual(value, this.getValue())
  }
}
```

3.4　常见问题解析

- v-on 可以绑定多个方法吗？

v-on 可以绑定多种类型的方法，可以是 click 事件，可以是 focus 事件，也可以是 change

事件，根据业务需求进行选择。但是，如果用 v-on 绑定了两个甚至多个 click 事件，那么 v-on 只会绑定第一个 click 事件，其他的会被自动忽略。

```
<input type="text" :value="name" @input="onInput" @focus="onFocus" @blur="onBlur" />
```

- 一个 Vue 实例可以绑定多个 element 元素吗？

这个疑问产生的原因是没有理解 el 的概念。el 为实例提供挂载元素，值可以是 CSS 选择符，或实际的 HTML 元素，或返回 HTML 元素的函数。注意，元素只用作挂载点。如果提供了模板，则元素被替换，除非 replace 为 false。元素可以用 vm.$el 访问。

用在 Vue.extend 中必须是函数值，这样所有实例不会共享元素。

如果在初始化时指定了这个选项，实例将立即进入编译过程；否则，需要调用 vm.$mount()，手动开始编译。

- 在 Vue 中如何让 v-for 循环出来的列表里面的 click 事件只对当前列表内元素有效？

比如模板如下，想点击 li 控制 span 的显隐：

```
<li @click="show">
  <span>1</span>
</li>
```

从数据角度出发，定义好数据结构，然后操作数据：

```
<body id="example">
  <div>
    <ul id="app">
      <li v-for='item in items' @click="toggle(item)">
        <span v-show='item.show'>{{item.content}}</span>
      </li>
    </ul>
  </div>
</body>
<script>
  new Vue({
    el: '#app',
    data: function() {
      return {
        items: [{
          content: '1 item',
```

```
          show: true
        }, {
          content: '2 item',
          show: true
        }, {
          content: '3 item',
          show: true
        }]
      }
    },
    methods: {
      toggle: function(item) {
            item.show = !item.show;
      }
    }
  })
</script>
```

或者通过$event 对象，获取当前事件源，然后操作下面的 span 元素，方法很多。

第 4 章

计算属性

通常我们会在模板中绑定表达式，模板是用来描述视图结构的。如果模板中的表达式存在过多的逻辑，模板会变得臃肿不堪，维护变得非常困难。因此，为了简化逻辑，当某个属性的值依赖于其他属性的值时，我们可以使用计算属性。

4.1 什么是计算属性

计算属性就是当其依赖属性的值发生变化时，这个属性的值会自动更新，与之相关的 DOM 部分也会同步自动更新。代码示例如下：

```
<div id="example">
  <input type="text" v-model="didi" />
  <input type="text" v-model="family" />
  <br>
  didi={{ didi }}, family={{ family }}, didiFamily = {{ didiFamily }}
</div>
var vm = new Vue({
  el: '#example',
  data: {
    didi: 'didi',
    family: 'family'
  },
  computed: {
    // 一个计算属性的 getter
    didiFamily: function () {
```

```
      // `this` 指向 vm 实例
      return this.didi + this.family
    }
  }
})
```

当 vm.didi 和 vm.family 的值发生变化时，vm.didiFamily 的值会自动更新，并且会自动同步更新 DOM 部分。

前面实例只提供了 getter，实际上除了 getter，我们还可以设置计算属性的 setter。代码示例如下：

```
var vm = new Vue({
  el: '#example',
  data: {
    didi: 'didi',
    family: 'family'
  },
  computed: {
    didiFamily: {
      // 一个计算属性的 getter
      get: function () {
        // `this` 指向 vm 实例
        return this.didi + ' ' + this.family
      },
      // 一个计算属性的 setter
      set: function (newVal) {
        var names = newVal.split(' ')
        this.didi = names[0]
        this.family = names[1]
      }
    }
  }
})
```

当设置 vm.didiFamily 的值时，vm.didi 和 vm.family 的值也会自动更新。

4.2　计算属性缓存

计算属性的特性的确很诱人，但是如果在计算属性方法中执行大量的耗时操作，则可能会带来一些性能问题。例如，在计算属性 getter 中循环一个大的数组以执行很多操作，那么当频繁调用该计算属性时，就会导致大量不必要的运算。

在 Vue.js 0.12.8 版本之前，只要读取相应的计算属性，对应的 getter 就会重新执行。而在 Vue.js 0.12.8 版本中，在这方面进行了优化，即只有计算属性依赖的属性值发生了改变时才会重新执行 getter。

这样也存在一个问题，就是只有 Vue 实例中被观察的数据属性发生了改变时才会重新执行 getter。但是有时候计算属性依赖实时的非观察数据属性。代码示例如下：

```
var vm = new Vue({
  data: {
    welcome: 'welcome to join didiFamily'
  },
  computed: {
    example: function () {
      return Date.now() + this.welcome
    }
  }
})
```

我们需要在每次访问 example 时都取得最新的时间而不是缓存的时间。从 Vue.js 0.12.11 版本开始，默认提供了缓存开关，在计算属性对象中指定 cache 字段来控制是否开启缓存。代码示例如下：

```
var vm = new Vue({
  data: {
    welcome: 'welcome to join didifamily'
  },
  computed: {
    example: {
      // 关闭缓存，默认为 true
      cache: false,
      get: function () {
        return Date.now() + this.welcome
      }
    }
  }
})
```

设置 cache 为 false 关闭缓存之后，每次直接访问 vm.example 时都会重新执行 getter 方法。

4.3 常见问题

在实际开发中使用计算属性时，我们会遇到各种各样的问题，以下是我们搜集到的一些常见问题以及解决方案。

4.3.1 计算属性 getter 不执行的场景

从前面章节中我们了解到，当计算属性依赖的数据属性发生改变时，计算属性的 getter 方法就会执行。但是在有些情况下，虽然依赖数据属性发生了改变，但计算属性的 getter 方法并不会执行。

当包含计算属性的节点被移除并且模板中其他地方没有再引用该属性时，那么对应的计算属性的 getter 方法不会执行。代码示例如下：

```
<div id="example">
  <button @click='toggleShow'>Toggle Show Total Price</button>
  <p v-if="showTotal">Total Price = {{totalPrice}}</p>
</div>

new Vue({
  el: '#example',
  data: {
    showTotal: true,
    basePrice: 100
  },
  computed: {
    totalPrice: function () {
      return this.basePrice + 1
    }
  },
  methods: {
    click: function() {
      this.showTotal = !this.showTotal
    }
  }
})
```

当点击按钮使 showTotal 为 false 时，此时 P 元素会被移除，在 P 元素内部的计算属性 totalPrice

的 getter 方法不会执行。但是当计算属性一直出现在模板中时，getter 方法还是会被执行。代码示例如下：

```
<div id="example">
  <button @click='toggleShow'>Toggle Show Total Price</button>
  <!-- 一直出现在模板中，不会条件性隐藏 -->
  <p>{{totalPrice}}</p>
  <p v-if="showTotal">Total Price = {{totalPrice}}</p>
</div>
```

4.3.2　在 v-repeat 中使用计算属性

有时候从后端获得 JSON 数据集合后，我们需要对单条数据应用计算属性。在 Vue.js 0.12 之前的版本中，我们可以在 v-repeat 所在元素上使用 v-component 指令。代码示例如下：

```
<div id="items">
  <p v-repeat="items" v-component="item">
    <button>{{fulltext}}</button>
  </p>
</div>

var items = [
  { number:1, text:'one' },
  { number:2, text:'two' }
]
var vue = new Vue({
  el: '#items',
  data: { items: items },
  components: {
    item: {
      computed: {
        fulltext: function() {
          return 'item ' + this.text
        }
      }
    }
  }
})
```

在 Vue.js 0.12.*版本中，Vue.js 废弃了 v-component 指令，所以我们需要使用自定义元素组

件来实现在 v-repeat 中使用计算属性。代码示例如下：

```
<div id="items">
  <my-item v-repeat="items" inline-template>
    <button>{{fulltext}}</button>
  </my-item>
</div>

var items = [
  { number:1, text:'one' },
  { number:2, text:'two' }
]
var vue = new Vue({
  el: '#items',
  data: { items: items },
  components: {
    'my-item': {
      replace: true,
      computed: {
        fulltext: function() {
          return 'item ' + this.text
        }
      }
    }
  }
})
```

第 5 章

表单控件绑定

在 Web 应用中，我们经常会使用表单向服务端提交一些数据，而通常也会在表单项中绑定一些如 input、change 等事件对用户输入的数据进行校验、更新等操作。在 Vue.js 中、我们可以使用 v-model 指令同步用户输入的数据到 Vue 实例 data 属性中，同时会对 radio、checkbox、select 等原生表单组件提供一些语法糖使表单操作更加容易。

5.1 基本用法

下面我们列举基本例子来看看如何使用 v-model 更新表单控件，具体的 v-model 数据绑定会在后续章节中展开介绍。

5.1.1 text

设置文本框 v-model 为 name，代码示例如下：

```
<span>Welcome {{ name }} join DDFE</span>
<br>
<input type="text" v-model="name" placeholder="join DDFE">
```

当用户操作文本框时，vm.name 会自动更新为用户输入的值，同时，span 内的内容也会随之改变。

5.1.2 checkbox

复选框 checkbox 在表单中会经常使用，下面我们来看看单个 checkbox 如何使用 v-model。

代码示例如下：

```
<input type="checkbox" id="checkbox" v-model="checked">
<label for="checkbox">{{ checked }}</label>
```

当用户勾选了 checkbox 时，vm.checked=true，否则 vm.checked=false，label 中的值也会随之改变。

大多数时候我们使用的都是多个复选框，即一个复选框组。此时，被选中的值将会放入一个数组中。代码示例如下：

```
<input type="checkbox" id="flash" value="flash" v-model="bizLines">
<label for="flash">快车</label>
<input type="checkbox" id="premium" value="premium" v-model="bizLines">
<label for="premium">专车</label>
<input type="checkbox" id="bus" value="bus" v-model="bizLines">
<label for="bus">巴士</label>
<br>
<span>Checked lines: {{ bizLines | json }}</span>

new Vue({
  el: '...',
  data: {
    bizLines: []
  }
})
```

5.1.3　radio

当单选钮被选中时，v-model 中的变量值会被赋值为对应的 value 值。代码示例如下：

```
<input type="radio" id="flash" value="flash" v-model="bizLine">
<label for="flash">快车</label>
<br>
<input type="radio" id="bus" value="bus" v-model="bizLine">
<label for="bus">巴士</label>
<br>
<span>Picked: {{ bizLine }}</span>
```

5.1.4　select

因为 select 控件分为单选和多选，所以 v-model 在 select 控件的单选和多选上会有不同的表

现。代码示例如下：

```
<select v-model="bizLine">
  <option selected value="flash">快车</option>
  <option value="premium">专车</option>
  <option value="bus">巴士</option>
</select>
<span>Selected: {{ bizLine }}</span>
```

当被选中的 option 有 value 属性时，vm.selected 为对应 option 的 value 值；否则为对应 option 的 text 值。

对于多选 select 控件，被选中的值会放入一个数组中。代码示例如下：

```
<select v-model="bizLines" multiple>
  <option selected value="flash">快车</option>
  <option value="premium">专车</option>
  <option value="bus">巴士</option>
</select>
<span>Selected: {{ bizLines | json }}</span>
```

我们也可以通过 v-for 指令来动态生成 option，v-for、v-bind 指令的具体用法请参阅指令部分。代码示例如下：

```
<select v-model="bizLine">
  <option v-for="option in options"  :value="option.value">
    {{ option.text }}
  </option>
</select>
<span>bizLine: {{ bizLine }}</span>

new Vue({
  el: '...',
  data: {
    bizLine: 'flash',
    options: [
      { text: '快车', value: 'flash' },
      { text: '专车', value: 'premium' },
      { text: '巴士', value: 'bus' }
    ]
  }
})
```

生成的 HTML 结构代码如下：

```
<select>
  <option value="flash">快车</option>
  <option value="premium">专车</option>
  <option value="bus">巴士</option>
</select>
```

5.2　值绑定

在通常情况下，对于 radio、checkbox、select 组件，通过 v-model 绑定的值都是字符串，checkbox 除外，checkbox 可能是布尔值。代码示例如下：

```
<!-- 勾选时 `picked` 的值是字符串 a -->
<input type="radio" v-model="picked" value="a">
<!-- 勾选时 `toggle` 的值是布尔值 true，否则是布尔值 false -->
<input type="checkbox" v-model="toggle">
<!-- 勾选时 `selected` 的值是字符串 abc-->
<select v-model="selected">
  <option value="abc">ABC</option>
</select>
```

有时我们会有动态绑定 Vue.js 实例属性的需求，这时可以使用 v-bind 来实现这个需求。通过 v-bind 来代替直接使用 value 属性，我们还可以绑定非字符串的值，如数值、对象、数组等。下面我们举例看看在各 form 表单中各控件如何使用该指令。

1. checkbox

```
<input
  type="checkbox"
  v-model="toggle"
  :true-value="a"
  :false-value="b">
```

- 勾选 checkbox 时，vm.toggle === vm.a。
- 未勾选 checkbox 时，vm.toggle === vm.b。

注：:true-value 和:false-value 只适合同一个 checkbox 组只有一个 checkbox 的情况。如果有多个 checkbox，请使用:value 进行值绑定。代码示例如下：

```
<input type="checkbox" id="flash" :value="flash" v-model="bizLines">
```

```
<label for="flash">{{ flash.name }}</label>
<input type="checkbox" id="premium" :value="premium" v-model="bizLines">
<label for="premium">{{ premium.name }}</label>
<input type="checkbox" id="bus" :value="bus" v-model="bizLines">
<label for="bus">{{ bus.name }}</label>
<br>
<span>Checked bizLines: {{ bizLines | json }}</span>

new Vue({
  el: '...',
  data: {
    flash: {name: '快车'},
    premium: {name: '专车'},
    bus: {name: '巴士'},
    bizLines: []
  }
})
```

2. radio

```
<input type="radio" v-model="pick" :value="a">
```

勾选 radio 时，vm.pick === vm.a。

3. select

```
<select v-model="selected">
  <option :value="{ number: 123 }">123</option>
</select>
```

用户勾选时，vm.selected === { number: 123}。

5.3　v-model 修饰指令

v-model 用来在视图与 Model 之间同步数据，但是有时候我们需要控制同步发生的时机，或者在数据同步到 Model 之前将数据转换为 Number 类型。我们可以在 v-model 指令所在的 form 控件上添加相应的修饰指令来实现这个需求。

5.3.1　lazy

在默认情况下，v-model 在 input 事件中同步输入框的值与数据，可以添加一个 lazy 特性，

从而改到在 change 事件中去同步。代码示例如下：

```
<input v-model="msg" lazy><br/>
{{msg}}
```

5.3.2　debounce

设置一个最小的延时，在每次敲击之后延时同步输入框的值到 Model 中。如果每次更新都要进行高耗操作（例如，在输入提示中 AJAX 请求）时，它较为有用。代码示例如下：

```
<input v-model="msg" debounce="500">
```

用户输入完毕 500ms 后，vm.msg 才会被更新。

注：该指令是用来延迟同步用户输入的数据到 Model 中，并不会延迟用户输入事件的执行。所以如果要想获取变化后的数据，我们应该用 vm.$watch()来监听 msg 的变化，而不是在事件中获取最新数据。要想延迟 DOM 事件的执行，请参阅过滤器章节中的 debounce 过滤器。

5.3.3　number

当传给后端的字段类型必须是数值的时候，我们可以在 v-model 所在控件上使用 number 指令，该指令会在用户输入被同步到 Model 中时将其转换为数值类型，如果转换结果为 NaN，则对应的 Model 值还是用户输入的原始值。代码示例如下：

```
<input v-model="age" number>
```

5.4　修饰指令原理

不少人应该都会对如何实现上面的修饰指令感兴趣，我们来看一下源码解析部分。

5.4.1　lazy 源码解析

所有的表单控件都绑定 change，只是在不设置 lazy 时，默认绑定 input。源码定义如下：

```
<!--源码目录：src/directives/public/model/text.js -->
var lazy = this.params.lazy
//. . .
this.on('change', this.rawListener)
if (!lazy) {
```

```
  this.on('input', this.listener)
}
```

5.4.2　debounce 源码解析

其实 debounce 和 filter 中的 debounce 原理相似，都是用的同一个函数，核心都是 setTimeout，只是 debounce 没有默认的 wait 值。源码定义如下：

```
<!--源码目录：src/directives/public/model/text.js -->
import {
  debounce as _debounce
} from '../../../util/index'

var debounce = this.params.debounce
//. . .
this.listener = this.rawListener = function () {
  //. . .
  self.set(val)
  //. . .
}
if (debounce) {
  this.listener = _debounce(this.listener, debounce)
}

<!--源码目录：src/util/lang.js -->
export function debounce (func, wait) {
  var timeout, args, context, timestamp, result
  var later = function () {
    var last = Date.now() - timestamp
    if (last < wait && last >= 0) {
      timeout = setTimeout(later, wait - last)
    } else {
      timeout = null
      result = func.apply(context, args)
      if (!timeout) context = args = null
    }
  }
  return function () {
    context = this
```

```
    args = arguments
    timestamp = Date.now()
    if (!timeout) {
      timeout = setTimeout(later, wait)
    }
    return result
  }
}
```

5.4.3　number 源码解析

调用 util 里面的 toNumber 方法。源码定义如下：

```
<!--源码目录：src/directives/public/model/text.js -->
import {
  toNumber
} from '../../../util/index'

var number = this.params.number
//. . .
this.listener = this.rawListener = function () {
  if (composing || !self._bound) {
    return
  }
  var val = number || isRange ? toNumber(el.value) : el.value
  self.set(val)
  //. . .
}


<!--源码目录：src/util/lang.js -->
export function toNumber (value) {
  if (typeof value !== 'string') {
    return value
  } else {
    var parsed = Number(value)
    return isNaN(parsed) ? value : parsed
  }
}
```

第 6 章

过滤器

在了解过滤器之前，我们需要明确一个概念——过滤器，本质上都是函数。其作用在于用户输入数据后，它能够进行处理，并返回一个数据结果。Vue.js 与 AngularJS 中的过滤器语法有些相似，使用管道符（|）进行连接。代码示例如下：

```
{{'abc' | uppercase}}
//'abc' => 'ABC'
```

这里使用了 Vue.js 内置的过滤器 uppercase，将字符串中的字母全部转换为大写形式。

Vue.js 支持在任何出现表达式的地方添加过滤器。除了上面例子中的 Mustache 风格（双大括号）的表达式之外，还可以在绑定指令的表达式后调用。代码示例如下：

```
<span v-text="message | uppercase"></span>
```

表达式的值可以根据用户的输入来动态改变，也可以像 abc 一样采用固定值。

过滤器可以接受参数，参数跟在过滤器名称后面，参数之间以空格分隔。代码示例如下：

```
{{ message | filterFunction 'arg1' arg2 }}
```

需要强调的是，过滤器函数将始终以表达式的值作为第一个参数。带引号的参数会被当作字符串处理，而不带引号的参数会被当作数据属性名来处理。这里，message 将作为第一个参数，字符串 arg1 作为第二个参数，表达式 arg2 的值在计算出来之后作为第三个参数传给过滤器。为避免混淆，下文中传入的参数个数及顺序只根据过滤器后跟的参数来统计。

熟悉 Linux shell 的读者可能对其中的管道符（|）的作用比较了解，即上一个命令的输出可以作为下一个命令的输入。Vue.js 过滤器中的管道符也同样支持这种方式的使用。这意味着 Vue.js 的过滤器支持链式调用，上一个过滤器的输出结果可以作为下一个过滤器的输入。代码

示例如下：

```
<span>{{ 'ddfe' | capitalize| reverse}}</span>
//->'ddfe' => 'Ddfe' => 'efdD'
//capitalize 过滤器：将输入字符串中的单词的首字母大写
//reverse 过滤器：反转字符串顺序
```

Vue.js 过滤器链式调用的特性能够让用户随心所欲地处理数据，这种将各种功能相对独立的过滤器函数组合起来解决复杂数据处理的方式与软件工程中的“高内聚、低耦合”设计思想有异曲同工之妙。结合后文将要介绍的 Vue.js 强大的自定义过滤器的功能，用户可以非常灵活地对数据进行处理，获得想要的数据形式。

6.1　内置过滤器

Vue.js 内置了一系列常用的过滤器，可以直接进行调用。这些内置过滤器都相对比较简单，如果要实现比较复杂或者需要定制的过滤功能，还是要借助自定义过滤器的方式。当然，这些内置的过滤器使用时无须定义，比较适合刚上手 Vue.js 的新人。我们来看一下 Vue.js 中常用的过滤器，如图 6-1 所示。

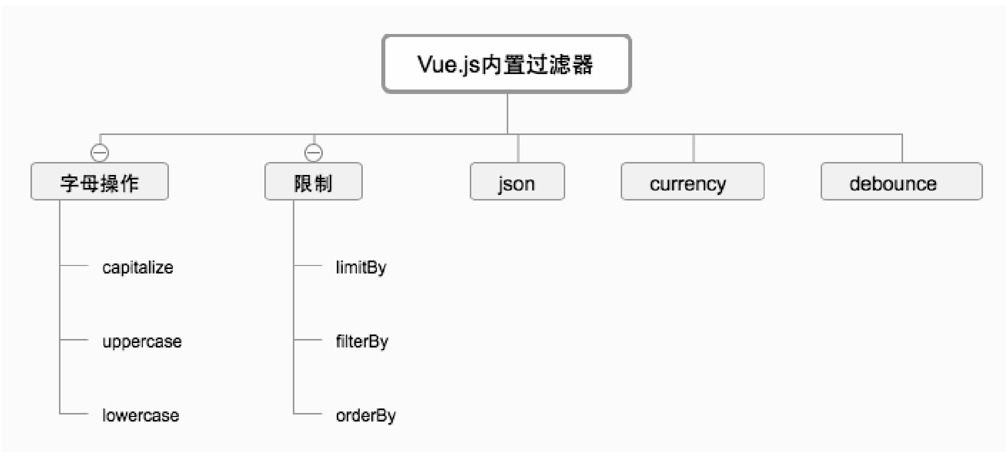


图6-1　Vue.js内置的常用过滤器

6.1.1　字母操作

Vue.js 内置了 capitalize、uppercase、lowercase 三个过滤器用于处理英文字符。注：这三个过滤器仅针对英文字符串使用。

1. capitalize

capitalize 过滤器用于将表达式中的首字母转换为大写形式。代码示例如下：

```
{{ 'ddfe' | capitalize }}
//'ddfe' => 'Ddfe'
```

2. uppercase

uppercase 过滤器用于将表达式中的所有字母转换为大写形式。代码示例如下：

```
{{ 'ddfe' | uppercase }}
// 'ddfe' => 'DDFE'
```

3. lowercase

lowercase 过滤器用于将表达式中的所有字母转换为小写形式。代码示例如下：

```
{{ 'DDFE' | lowercase }}
// 'DDFE' => 'ddfe'
```

6.1.2 json 过滤器

Vue.js 中的 json 过滤器本质上是 JSON.stringify()的精简缩略版，可将表达式的值转换为 JSON 字符串，即输出表达式经过 JSON.stringify()处理后的结果。json 可接受一个类型为 Number 的参数，用于决定转换后的 JSON 字符串的缩进距离，如果不输入该参数，则默认为 2。代码示例如下：

```
<pre>{{ didiFamily | json 4 }}</pre>
/*
以四个空格的缩进打印一个对象：
didiFamily: { 'name': 'ddfe', 'age': 3}
=>

{
  'name': 'ddfe',
  'age': 3
}
*/
```

6.1.3 限制

Vue.js 中内置了 limitBy、filterBy、orderBy 三个过滤器用于处理并返回过滤后的数组，比

如与 v-for 搭配使用。注意，这三个过滤器所处理的表达式的值必须是数组，否则程序会报错。

1. limitBy

limitBy 过滤器的作用是限制数组为开始的前 *N* 个元素，其中 *N* 由传入的第一个参数指定。第二个参数可选，用于指定开始的偏移量，默认为 0，即不偏移。如果第二个参数为 5，则表示从数组下标为 5 的地方开始计数。代码示例如下：

```
<!-- 只显示开始的 10 个元素 -->
<div v-for="item in items | limitBy 10"></div>

<!-- 显示第 5 到 15 个元素-->
<div v-for="item in items | limitBy 10 5"></div>
```

2. filterBy

filterBy 过滤器的使用比较灵活，其第一个参数可以是字符串或者函数。过滤条件是：'string || function' + in + 'optionKeyName'。

如果第一个参数是字符串，那么将在每个数组元素中搜索它，并返回包含该字符串的元素组成的数组。代码示例如下：

```
< div v-for="item in items | filterBy 'hello'"></div>
```

上例中，只显示包含 hello 字符串的元素。

如果 item 是一个对象，过滤器将递归地在它所有的属性中搜索。为了缩小搜索范围，可以指定一个搜索字段。代码示例如下：

```
<div v-for="member in didiFamily | filterBy 'ddfe' in 'name'"></div>
```

上例中，过滤器只在用户对象的 name 属性中搜索 ddfe。最好始终限制搜索范围以提高效率与性能。

也可以在多个字段中进行搜索，字段与字段之间以空格分隔。代码示例如下：

```
<li v-for="user in users | filterBy 'Chris' in 'name' 'nickname'"></li>
```

还可以将搜索字段存放在一个数组中，这样当修改搜索字段时只需修改数组即可，无须再修改 View 层。代码示例如下：

```
<!-- fields = ['fieldA', 'fieldB'] -->
<div v-for="user in users | filterBy searchText in fields"></div>
```

上面的例子中均使用了静态参数，当然也可以使用动态参数作为搜索目标或搜索字段。结

合 v-model，我们可以轻松地实现输入提示效果。代码示例如下：

```
<div id="dynamic-filter-by">
  <input v-model="name">
  <ul>
    <li v-for="user in users | filterBy name in 'name'">
      {{ user.name }}
    </li>
  </ul>
</div>

new Vue({
  el: '#dynamic-filter-by',
  data: {
    name: '',
    users: [
      { name: 'Bruce' },
      { name: 'Chuck' },
      { name: 'Jackie' }
    ]
  }
})
```

上例中，根据输入框中用户输入的数据，可以实时过滤出包含用户输入的字符串的数组元素，十分高效、简洁。动态参数作为搜索字段的方式与此类似，不再赘述。

如果 filterBy 的第一个参数是函数，则过滤器将根据函数的返回结果进行过滤。此时 filterBy 过滤器将调用 JavaScript 数组中内置的函数 filter()对数组进行处理，待过滤数组中的每个元素都将作为参数输入并执行传入 filterBy 中的函数。只有函数返回结果为 true 的数组元素才符合条件并将存入一个新的数组，最终返回结果即为这个新的数组。

3. orderBy

orderBy 过滤器的作用是返回排序后的数组。过滤条件是：'string || array ||function' + 'order ≥0 为升序 || order < 0 为降序'。第一个参数可以是字符串、数组或者函数。第二个参数 order 可选，决定结果为升序或降序排列，默认为 1，即升序排列。

若输入参数为字符串，则可同时传入多个字符串作为排序键名，字符串之间以空格分隔。

代码示例如下：

```
<ul>
  <li v-for="user in users | orderBy 'lastName' 'firstName' 'age'">
    {{ user.lastName }} {{ user.firstName }} {{ user.age }}
  </li>
</ul>
```

此时将按照传入的排序键名的先后顺序进行排序。

也可以将排序键名按照顺序放入一个数组中，然后传入一个数组参数给 orderBy 过滤器即可。代码示例如下：

```
<!-- sortKey = ['lastName','firstName','age']-->
<ul>
  <li v-for="user in users | orderBy sortKey">
    {{ user.lastName }} {{ user.firstName }} {{ user.age }}
  </li>
</ul>
```

当传入第一个参数为函数时，orderBy 过滤器与 JavaScript 数组中内置的 sort()函数表现一致。

注：事实上，当传入参数为字符串或者数组时，最终调用的也是 sort()函数，只不过 Vue.js 提前作了一些处理，比如设置了默认的 compare 函数等，根据传入的 compare 函数进行排序。

6.1.4　currency 过滤器

currency 过滤器的作用是将数字值转换为货币形式输出。其第一个参数接受类型为 String 的货币符号，如果不输入，则默认为美元符号$。第二个参数接受类型为 Number 的小数位，如果不输入，则默认为 2。注意，如果第一个参数采取默认形式，而需要第二个参数修改小数位，则第一个参数不可省略。代码示例如下：

```
{{ amount | currency }}
// 12345 => $12,345.00
```

使用其他符号，比如英镑符号，代码示例如下：

```
{{ amount | currency '£' }}
// 12345 => £12, 345.00
```

将小数位调整为 3 位，代码示例如下：

```
{{ amount | currency '$' 3}}
// 12345 => $12,345.000
```

6.1.5　debounce 过滤器

debounce 过滤器的作用是延迟处理器一定的时间执行。其接受的表达式的值必须为函数，因此其一般与 v-on 等指令结合使用。debounce 接受一个可选的参数作为延迟时间，单位为毫秒。如果没有该参数，则默认的延迟时间为 300 毫秒。经过 debounce 包装的处理器在调用之后将至少延迟设定的时间再执行。如果在延迟结束前再次调用，则延迟时长将重置为设定的时间。通常，在监听用户 input 事件时使用 debounce 过滤器比较有用，可以防止频繁调用方法。debounce 的用法参考如下：

```
<input @keyup="onKeyup | debounce 500">
```

6.2　自定义过滤器

大多数情况下，Vue.js 中内置的过滤器并不能满足我们的需求，好在 Vue.js 还提供了自定义过滤器的 API 供用户进行功能扩展。在学习 Vue.js 自定义过滤器之前，我们先来看看如何在 AngularJS 中自定义过滤器。代码示例如下：

```
angular.module('dd.filters', [])
    .filter('reverse', function (value) {
        return value.split('').reverse().join('')
})

<p>{{ msg | capitalize | reverse: '123' }}</p>
<!--此处 reverse 过滤器带的参数'123'并不起作用，只是用于展示 AngularJS 的过滤器接受参数的书写形式 -->
```

Vue.js 中自定义过滤器的语法与 AngularJS 在形式上相近，但是语法略有不同。

6.2.1　fillter 语法

在 Vue.js 中也存在一个全局函数 Vue.filter 用于构造过滤器：

```
Vue.filter(ID,function(){})
```

该函数接受两个参数，其中第一个参数为过滤器 ID，作为用户自定义过滤器的唯一标识；第二个参数则为具体的过滤器函数。过滤器函数以值为参数，返回转换后的值。

1. 单个参数

注册一个名为 reverse 的过滤器，作用是将字符串反转输出。代码示例如下：

```
Vue.filter('reverse', function (value) {
  return value.split('').reverse().join('');
})

<span v-text="message | reverse"></span>
<!-- 'abc' => 'cba' -->
```

2. 多参数

过滤器函数除了以值为参数外，还支持接受任意数量的参数，参数之间以空格分隔。代码示例如下：

```
Vue.filter('wrap', function (value, begin, end) {
  return begin + value + end
})

<span v-text="message | wrap 'before' 'after'"></span>
<!-- 'hello' => 'before hello after' -->
```

3. 双向过滤器

上面的过滤器函数都是在 Model 数据输出到 View 层之前进行数据转化的，实际上 Vue.js 还支持把来自视图（input 元素）的值在写回模型前进行转化，即双向过滤器。代码示例如下：

```
Vue.filter(id, {
    // model -> view
    // read 函数可选
    read: function(val){},

    // view -> model
    // write 函数将在数据被写入 Model 之前调用
    // 两个参数分别为表达式的新值和旧值
    write: function(newVal, oldVal){}
});
```

4. 动态参数

filter 语法还有一个需要注意的点：动态参数。如果过滤器参数没有用引号包起来，则它会在当前 vm 作用域内动态计算。此外，过滤器函数的 this 始终指向调用它的 vm。代码示例如下：

```
<input v-model="userInput">
<span>{{msg | concat userInput}}</span>

<!--此处过滤器接受的参数 userInput 根据用户输入动态计算-->
Vue.filter('concat', function (value, input) {
  // `input` === `this.userInput`
  return value + input
})
```

6.2.2　教你写一个 filter

针对常规过滤器，6.2.1 节中已经给出一个比较简单的过滤器 reverse 的实现，代码示例如下：

```
Vue.filter('reverse', function (value) {
  return value.split('').reverse().join('');
})
```

需要注意两点：

- 需要给定过滤器一个唯一标识。如果用户自定义的过滤器和 Vue.js 内置的过滤器冲突，那么 Vue.js 内置的过滤器将会被覆盖；如果后注册的过滤器和之前的过滤器冲突，则之前注册的过滤器层被覆盖。
- 过滤器函数的作用是输入表达式的值，经过处理后输出。因此，定义的函数最好可以返回有意义的值。函数没有 return 语句不会报错，但这样的过滤器没有意义。

对于双向过滤器，这里给出一个例子供参考，代码示例如下：

```
<div id="example">
  <p>{{ message }}</p>
  <input type='text' v-model="message | filterExample"
</div>
Vue.filter('filterExample', {
    read: function(val){
        return 'read ' + val;
    },
    write: function(newVal, oldVal){
        return oldVal + ' write';
    }
});

var demo = new Vue({
```

```
    el: '#example',
    data: {
        message: 'hello world'
    }
});
```

在初始情况下，页面显示如图 6-2 所示。message 表达式的值经过 filterExample 中的 read 函数处理，输出到 View 层。当我们在 input 框中修改 message 的值时，filterExample 中的 write 函数将在数据输出到 Model 层之前处理，这里将返回 message 的旧值+ 'write'，然后输出到 Model 层，因此 message 的值变更为'hello world write'并显示在页面上，如图 6-3 所示。



图6-2　初始情况

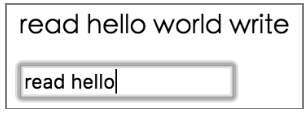


图6-3　修改input框中message的值

6.3　源码解析

6.3.1　管道实现

为了让过滤器的调用尽可能简单并且支持链式调用等特性，Vue.js 背后做了不少工作。通过查看源码可以知道，管道的实现主要借助于几个函数，如图 6-4 所示。

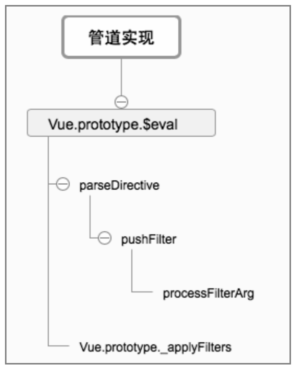


图6-4　管道实现结构图

管道实现的原理是：Vue.prototype.$eval 接受类型为 String 的后跟过滤器及其参数的表达式（例如'message | filterExample args1'），运用正则表达式检测传入的字符串中是否存在管道符(|)，如果存在，则调用 parseDirective 函数将传入的字符串整理输出为一个对象。例如，如果传入的表达式为'a + 1 | uppercase'，经过 parseDirective 处理后将输出为如下形式：

```
{
  expression: 'a + 1',
  filters: [
    { name: 'uppercase', args: null }
  ]
}
```

其中，parseDirective 调用了 pushFilter 和 processFilterArg 两个函数。前者的作用是将一个过滤器函数加入对象内的 filters 数组中；后者的作用是检查一个参数是否为动态参数，并且去

掉静态参数上的引号。

在得到 parseDirective 处理好的对象之后，Vue.prototype.$eval 将调用 Vue.prototype._applyFilters 函数，该函数将表达式的值作为参数输入并依次调用 filters 数组中的过滤器函数，最后输出经过所有过滤器处理的数据结果，达到链式调用过滤器的效果。

以上函数在 Vue.js 源码中均可找到，在此不再贴出详细的实现方式。

6.3.2 过滤器解析

Vue.js 内置的过滤器在函数实现上都比较简单，读者应该也能够很轻松地实现相应的功能。这里仅节选几个过滤器的实现源码，不排除有性能更优、复杂度更低的实现，读者可自行设计编写。

1. capitalize

```
function capitalize (value) {
  if (!value && value !== 0) return ''
  value = value.toString()
  return value.charAt(0).toUpperCase() + value.slice(1)
}
```

2. uppercase

```
function uppercase (value) {
  return (value || value === 0)
  ? value.toString().toUpperCase()
  : ''
}
```

3. lowercase

```
function lowercase (value) {
  return (value || value === 0)
  ? value.toString().toLowerCase()
  : ''
}
```

6.4 常见问题解析

1. filterBy/orderBy 过滤后$index 的索引

在使用 filterBy 或者 orderBy 对表达式进行过滤时，如果同时需要将$index 作为参数，此时

的$index 将会根据表达式数组或对象过滤后的值进行索引。代码示例如下：

```
<ul id="example">
  <li v-for="item in items | orderBy 'age' ">
    {{ item.message }}-{{$index}}
  </li>
</ul>

var example = new Vue({
  el: '#example',
  data: {
    items: [
      { message: '顺风车',age: 1 },
      { message: '出租车',age: 10 },
      { message: '快车',age: 6 }
    ]
  }
})
// 最终显示顺序为：顺风车-0、快车-1、出租车-2
```

2. 自定义 filter 的书写位置

自定义 filter 可以写在全局的 Vue 下，代码示例如下：

```
Vue.filter('reverse', function (value) {
  return value.split('').reverse().join('');
})
```

也可以写在实例当中，代码示例如下：

```
var demo = new Vue({
  el:'#demo',
  data:{},
  filters:{
    // 自定义 filter 事件的位置
    reverse:function(value){
      return value.split('').reverse().join('');
    }
  },
  methods:{}
})
```

二者本质上并无区别，可选择一种使用。但是采用 Vue.filter 时，需要在实例化 Vue 对象前定义，否则自定义的 filter 将不起作用。

第 7 章

Class 与 Style 绑定

对于数据绑定，一个常见的需求是操作元素的 class 列表和它的内联样式。因为它们都是 attribute，我们可以用 v-bind 处理它们：只需要计算出表达式最终的字符串。不过，字符串拼接麻烦又易错。因此，在 v-bind 用于 class 和 style 时，Vue.js 专门增强了它。表达式的结果类型除了字符串以外，还可以是对象或数组。

7.1 绑定 HTML Class

7.1.1 对象语法

我们可以传给 v-bind:class 一个对象，以动态地切换 class。注意，v-bind:class 指令可以与普通的 class 特性共存。代码示例如下：

```
<div id='example' class="static" v-bind:class="{ 'didi-orange': isRipe, 'didi-green':
isNotRipe }"></div>
var vm = new Vue({
  el: 'example',
  data: {
    isRipe: true,
    isNotRipe: false
  }
})
```

渲染为：

```
<div id='example' class="static didi-orange"></div>
```

当 isRipe 和 isNotRipe 变化时，class 列表将相应地更新。例如，如果 isNotRipe 变为 true，

那么 class 列表将变为"static didi-orange didi-green"。（当然，一般情况下，v-bind:class 绑定的对象中只有一个 class 会生效，这取决于用户自己的设置。）

注：尽管可以用 Mustache 标签绑定 class，比如 class="{{ className }}"，但是我们不推荐这种写法和 v-bind:class 混用。

我们也可以直接绑定数据中的一个对象，代码示例如下：

```
<div id='example' v-bind:class="ddfe"></div>
var vm = new Vue({
  el: 'example',
  data: {
    ddfe: {
      'didi-orange': true,
      'didi-green': false
    }
  }
})
```

还可以在这里绑定一个返回对象的计算属性。这是一种常用且强大的模式。代码示例如下：

```
<div id='example' v-bind:class="ddfe"></div>
var vm = new Vue({
  el: 'example',
  data: {
    didiAge:4,
    didiMember:6000
  }
  computed: {
    ddfe: function(){
      return {
        'didi-orange': this.didiAge>3 ? true: false,
        'didi-large': this.didiMember>1000 ? true: false
      }
    }
  }
})
```

7.1.2　数组语法

我们可以把一个数组传给 v-bind:class，以应用一个 class 列表。代码示例如下：

```
<div id='example' v-bind:class="[didiHandsome, didiBeautiful]">
var vm = new Vue({
  el: 'example',
  data: {
    didiHandsome: 'didi-handsome',
    didiBeautiful: 'didi-beautiful'
  }
})
```

渲染为：

```
<div id='example' class="didi-handsome didi-beautiful"></div>
```

如果想根据条件切换列表中的 class，则可以用三元表达式。代码示例如下：

```
<div id='example' v-bind:class="[didiHandsome, isRipe ? didiOrange: '']">
```

此例始终添加 didiHandsome，但是只有在 isRipe 为 true 时才会添加 didiOrange。

不过，当有多个条件 class 时这样写有些烦琐。在 Vue.js 1.0.19 及以后版本中，可以在数组语法中使用对象语法。代码示例如下：

```
<div id='example' v-bind:class="[didiHandsome, { didiOrange: isRipe, didiGreen: isNotRipe }]">
```

7.2　绑定内联样式

7.2.1　对象语法

v-bind:style 的对象语法十分直观——看着非常像 CSS，其实它是一个 JavaScript 对象。CSS 属性名可以用驼峰式（camelCase）或短横分隔命名（kebab-case）。代码示例如下：

```
<div id='example' v-bind:style="{ color: didiColor, fontSize: fontSize + 'px' }"></div>
var vm = new Vue({
  el: 'example',
  data: {
    didiColor: 'orange',
```

```
    fontSize: 30
  }
})
```

通常直接绑定到一个样式对象更好，让模板更清晰。代码示例如下：

```
<div id='example' v-bind:style="ddfe"></div>
var vm = new Vue({
  el: 'example',
  data: {
    ddfe: {
      color: orange,
      fontSize: '13px'
    }
  }
})
```

同样的，对象语法常常结合返回对象的计算属性使用。代码示例如下：

```
<div id='example' v-bind:style="ddfe"></div>
var vm = new Vue({
  el: 'example',
  data: {
  didiAge:4,
    didiMember:6000
}
  computed: {
    ddfe: function(){
    return {
      color: this.didiAge>3 ? orange: green,
      fontSize: this.didiMember>1000 ? 20px: 10px
    }
    }
  }
})
```

7.2.2　数组语法

v-bind:style 的数组语法可以将多个样式对象应用到一个元素上。代码示例如下：

```
<div v-bind:style="[ddfe, didiFamily]">
```

7.2.3 自动添加前缀

当 v-bind:style 使用需要厂商前缀的 CSS 属性时，如 transform，Vue.js 会自动侦测并添加相应的前缀。在 Vue.js 源码中采用 prefix 函数来完成这个功能，源码定义如下：

```
const prefixes = ['-webkit-', '-moz-', '-ms-']
const camelPrefixes = ['Webkit', 'Moz', 'ms']
function prefix (prop) {
  // hyphenate 函数的功能是将驼峰式（camelCase）属性转换为短横分隔式（kebab-case）
  prop = hyphenate(prop)
  // camelize 函数的功能是将短横分隔式（kebab-case）属性转换为驼峰式（camelCase）
  var camel = camelize(prop)
  var upper = camel.charAt(0).toUpperCase() + camel.slice(1)
  if (!testEl) {
    testEl = document.createElement('div')
  }
  var i = prefixes.length
  var prefixed
  if (camel !== 'filter' && (camel in testEl.style)) {
    return {
      kebab: prop,
      camel: camel
    }
  }
  while (i--) {
    prefixed = camelPrefixes[i] + upper
    if (prefixed in testEl.style) {
      return {
        kebab: prefixes[i] + prop,
        camel: prefixed
      }
    }
  }
}
```

该函数的基本原理是检测输入的属性是否在浏览器的 element.style 中存在，如果存在则不作变动；如果不存在，则为其添加浏览器私有前缀。在 Vue.js 中 prefix 函数的实现借鉴了著名

的 paulirish 中的 gimmePrefix 函数的实现方式：

```
function gimmePrefix(prop){
  var prefixes = ['Moz','Khtml','Webkit','O','ms'],
    elem= document.createElement('div'),
    upper = prop.charAt(0).toUpperCase() + prop.slice(1);
  if (prop in elem.style)
    return prop;
  for (var len = prefixes.length; len--; ){
    if ((prefixes[len] + upper)  in elem.style)
    return (prefixes[len] + upper);
  }
  return false;
}
```

二者的区别在于 Vue.js 中的 prefix 函数用 while 循环代替了 for 循环，同时 testEl 的命名更加友好。

需要注意的是，对于"filter"属性，Vue.js 作者表示这是 Chrome 浏览器的一个 bug：Chrome 只支持带 webkit 前缀的版本即-webkit-filter，但是在 Chrome 的 element-style 中却含有"filter"这个无前缀属性，因此，根据 prefix 函数的原理，Vue.js 在这里不会为 filter 添加前缀，导致该属性失效。为了修复该 bug，在 prefix 函数中，如果检测到属性是 filter，则也为其添加前缀。

此外，一般来说，浏览器私有前缀包括 Firefox 的-moz-、IE 的-ms-、Safari 和 Chrome 的-webkit-，以及 Opera 的-o-，但是 Vue.js 只添加了前面三种前缀，这是读者需要注意的一点。

第 8 章

过渡

过渡效果在交互体验中的重要性不言而喻。以往我们使用 jQuery 添加或移除元素的类，搭配 CSS 中定义好的样式，再引用一些 JavaScript 库之后，可以做出非常复杂、惊艳的动态效果，不过这一套方法仍略显烦琐。Vue.js 内置了一套过渡系统，可以在元素从 DOM 中插入或移除时自动应用过渡效果。Vue.js 会在适当的时机触发 CSS 过渡或动画，用户也可以提供相应的 JavaScript 钩子函数在过渡过程中执行自定义 DOM 操作。

应用过渡效果，需要在目标元素上使用 transition 特性。代码示例如下：

```
<div v-if="show" transition="my-transition"></div>
```

transition 特性可以与以下资源一起搭配使用：

- v-if
- v-show
- v-for（只在插入和删除时触发，使用 vue-animated-list 插件）
- 动态组件（见“组件”一章）
- 在组件的根节点上，并且被 Vue 实例的 DOM 方法（如 vm.$appendTo(el)）触发

当插入或者删除带有 transition 特性的元素时，Vue.js 将执行以下操作：

- 尝试以 ID“my-transition”查找 JavaScript 过渡钩子对象，该对象通过 Vue.transition(id, hooks)或 transitions 选项注册（后文将介绍）。如果找到了，将在过渡的不同阶段调用相应的钩子。

- 自动嗅探目标元素是否有 CSS 过渡或动画（按照 Vue.js 指定的方式添加类名即可），并在合适时添加/删除 CSS 类名，免去了用户自己进行相关操作的烦琐。
- 如果没有找到 JavaScript 钩子并且也没有检测到 CSS 过渡/动画，DOM 操作（插入/删除）将在下一帧中立即执行。

8.1　CSS 过渡

Vue.js 为用户定义了一套规则用于很方便地启用 CSS 过渡，典型的 CSS 过渡代码示例如下：

```
<div v-if="show" transition="expand">hello</div>
```

然后为.expand-transition、.expand-enter 和 .expand-leave 添加 CSS 规则，这样 Vue.js 就会在相应的阶段检测相应的 CSS 类的存在并及时添加和删除。添加 CSS 样式的代码示例如下：

```
/* 必需 */
.expand-transition {
  transition: all .3s ease;
  height: 30px;
  padding: 10px;
  background-color: #eee;
  overflow: hidden;
}

/* .expand-enter 定义进入的开始状态 */
/* .expand-leave 定义离开的结束状态 */
.expand-enter, .expand-leave {
  height: 0;
  padding: 0 10px;
  opacity: 0;
}
```

可以在同一个元素上通过动态绑定实现不同的过渡，代码示例如下：

```
<div v-if="show" :transition="transitionName">hello</div>
new Vue({
  el: '...',
  data: {
    show: false,
```

```
    transitionName: 'fade'
  }
})
```

除此之外，还可以提供 JavaScript 钩子函数，以下为简单示例，其具体语法见 8.2 节。

```
Vue.transition('expand', {
  beforeEnter: function (el) {
    el.textContent = 'beforeEnter'
  },
  enter: function (el) {
    el.textContent = 'enter'
  },
  afterEnter: function (el) {
    el.textContent = 'afterEnter'
  },
  enterCancelled: function (el) {
  },
  beforeLeave: function (el) {
    el.textContent = 'beforeLeave'
  },
  leave: function (el) {
    el.textContent = 'leave'
  },
  afterLeave: function (el) {
    el.textContent = 'afterLeave'
  },
  leaveCancelled: function (el) {
  }
})
```

8.1.1 内置 Class 类名

类名的添加以及切换取决于 transition 特性的值，例如 transition = 'boom'，会有三个内置类名：

- .boom-transition，始终保留在元素上。
- .boom-enter，定义进入过渡的开始状态。只应用一帧，然后立即删除。
- .boom-leave，定义离开过渡的结束状态。在离开过渡开始时生效，在它结束后删除。

值得注意的是，如果 transition 没有指定值，即 id 为空，则使用默认类名：.v-transition、.v-enter、.v-leave。

8.1.2　自定义 CSS 类名

用户可以在过渡的 JavaScript 中声明自定义的 CSS 过渡类名。这些自定义的类名会覆盖默认的类名。当需要和第三方的 CSS 动画库如 Animate.css 配合时会非常有用。代码示例如下：

```
<div v-show="ok" class="animated" transition="bounce">Watch me bounce</div>
Vue.transition('bounce', {
  enterClass: 'bounceInLeft',
  leaveClass: 'bounceOutRight'
})
```

注：此特性在 Vue.js 1.0.14 版本中是新增的。

8.1.3　显式声明 CSS 过渡类型

Vue.js 需要给过渡元素添加事件侦听器来侦听过渡何时结束。基于所使用的 CSS，该事件要么是 transitionend，要么是 animationend。如果用户只使用了两者中的一种，那么 Vue.js 将能够根据生效的 CSS 规则自动推测出对应的事件类型。但是，在有些情况下，一个元素可能需要同时带有两种类型的动画。比如用户可能希望让 Vue.js 来触发一个 CSS 动画，同时该元素在鼠标悬浮时又有 CSS 过渡效果。在这样的情况下，用户需要显式地声明希望 Vue.js 处理的动画类型（animation 或 transition），代码示例如下：

```
Vue.transition('bounce', {
  // 该过渡效果将只侦听 `animationend` 事件
  type: 'animation'
})
```

注：此特性在 Vue.js 1.0.14 版本中是新增的。

8.1.4　动画案例

CSS 动画的用法同 CSS 过渡，区别是在动画中 v-enter 类名在节点插入 DOM 后不会立即删除，而是在 animationend 事件触发时删除。如果要运用 JavaScript 过渡中的钩子函数，正如上一节所述，需要显式地将传入的 hooks 对象中的 type 属性设置为 animation。代码示例如下：

```
<!--为简便起见，省略了 animation 和 transform 等兼容性前缀-->
<span v-show="show" transition="bounce">Look at me!</span>
```

```
.bounce-transition {
  display: inline-block; /* 否则 scale 动画不起作用 */
}
.bounce-enter {
  animation: bounce-in .5s;
}
.bounce-leave {
  animation: bounce-out .5s;
}
@keyframes bounce-in {
  0% {
    transform: scale(0);
  }
  50% {
    transform: scale(1.5);
  }
  100% {
    transform: scale(1);
  }
}
@keyframes bounce-out {
  0% {
    transform: scale(1);
  }
  50% {
    transform: scale(1.5);
  }
  100% {
    transform: scale(0);
  }
}
```

8.1.5　过渡流程

我们以如下的结构为例来说明 Vue.js 内部是如何处理过渡的：

```
<div v-show="show" transition="">Transition example</div>
```

当 show 属性改变时，Vue.js 将相应地插入或删除<div>元素，按照如下规则改变过渡的 CSS

类名（以下每一项在用户没有设置时都将会跳过）：

- 如果 show 变为 false，Vue.js 将：
  - 调用 beforeLeave 钩子；
  - 将 v-leave 类名添加到元素上以触发过渡；
  - 调用 leave 钩子；
  - 等待过渡结束（监听 transitionend 事件）；
  - 从 DOM 中删除元素并删除 v-leave 类名；
  - 调用 afterLeave 钩子。
- 如果 show 变为 true，Vue.js 将：
  - 调用 beforeEnter 钩子；
  - 将 v-enter 类名添加到元素上；
  - 把它插入 DOM 中；
  - 调用 enter 钩子；
  - 强制一次 CSS 布局，让 v-enter 确实生效。然后删除 v-enter 类名，以触发过渡，回到元素的原始状态；
  - 等待过渡结束；
  - 调用 afterEnter 钩子。

另外，如果进入过渡还在进行中时删除元素，将调用 enterCancelled 钩子，以清理变动或 enter 创建的计时器；反之，对于离开过渡也是如此。

上面所有的钩子函数在调用时，它们的 this 均指向其所属的 Vue 实例。编译规则为：过渡在哪个上下文中编译，它的 this 就指向哪个上下文。

最后，enter 和 leave 可以有第二个可选的回调参数，用于显式控制过渡如何结束。因此不必等待 CSS transitionend 事件，Vue.js 将等待用户手工调用这个回调函数，以结束过渡。代码示

例如下：

```
enter: function (el) {
  // 没有第二个参数
  // 由 CSS transitionend 事件决定过渡何时结束
}

enter: function (el, done) {
  // 有第二个参数
  // 过渡只有在调用 `done` 时结束
}
```

注：当多个元素一起过渡时，Vue.js 会批量处理，只强制一次布局。

8.2　JavaScript 过渡

也可以只使用 JavaScript 钩子，不用定义任何 CSS 规则。当只使用 JavaScript 过渡时，enter 和 leave 钩子需要调用 done 回调，否则它们将被同步调用，过渡将立即结束。

建议只使用 JavaScript 钩子时，为 JavaScript 过渡显式声明 css: false，Vue.js 将跳过 CSS 检测。这样也会防止 CSS 规则对过渡的干扰。

我们使用 jQuery 来注册一个自定义的 JavaScript 过渡，代码示例如下：

```
Vue.transition('fade', {
  css: false,
  enter: function (el, done) {
    // 元素已被插入 DOM 中
    // 在动画结束后调用 done
    $(el)
      .css('opacity', 0)
      .animate({ opacity: 1 }, 1000, done)
  },
  enterCancelled: function (el) {
    $(el).stop()
  },
  leave: function (el, done) {
    // 与 enter 相同
    $(el).animate({ opacity: 0 }, 1000, done)
  },
  leaveCancelled: function (el) {
```

```
    $(el).stop()
  }
})
```

然后在 transition 特性中声明，代码示例如下：

```
<p transition="fade"></p>
```

完整地注册一个 JavaScript 过渡的代码示例如下：

```
Vue.transition(ID, {
  enter: function () {},
  leave: function () {}
})
```

Vue.transition 方法接受两个参数，其中第一个参数是过渡 ID，作为用户自定义的 transition 的唯一标识；第二个参数是一个对象 hooks。hooks 必须含有 enter 和 leave 两个类型为 function 的属性。这便是最基本的一个 JavaScript 过渡。除此之外，hooks 还可以包含其他属性，代码示例如下：

```
{
  type: 'animation'
  css: true,
  enterClass: 'bounceInLeft',
  leaveClass: 'bounceOutRight'
  beforeEnter: function (el) {},
  enter: function (el) {},
  afterEnter: function (el) {},
  enterCancelled: function (el) {},
  beforeLeave: function (el) {},
  leave: function (el) {},
  afterLeave: function (el) {},
  leaveCancelled: function (el) {},
  stagger: function(index){}
}
```

注：以上只是列出 hooks 对象可以设置的属性，不代表所有属性都需要被设置，属性的值也仅供参考。而且有些属性存在互斥关系。比如设置了 css 值为 false 时，enterClass 与 leaveClass 的设置将无效。各属性的含义可见其他各节，或者查阅 Vue.js 官方文档。

8.3　渐进过渡

transition 与 v-for 一起使用时可以创建渐进过渡，即让 v-for 中的每个过渡项目可以依次产

生过渡效果，而不是一次性同步产生过渡效果。给过渡元素添加一个特性 stagger、enter-stagger 或 leave-stagger（以毫秒作为单位），分别可以控制每个过渡项目的延迟时间、进入时的延迟时间以及离开时的延迟时间。我们来看看如何为列表元素添加渐进过渡效果。代码示例如下：

```
<div v-for="item in list" transition="myStaggeredTransition" stagger="100"></div>
.myStaggeredTransition-transition {
  transition: all .5s ease;
  overflow: hidden;
  margin: 0;
  height: 20px;
}
.myStaggeredTransition-enter, .myStaggeredTransition-leave {
  opacity: 0;
  height: 0;
}
```

或者提供一个钩子 stagger、enter-stagger 或 leave-stagger，以更好地控制。代码示例如下：

```
Vue.transition('myStaggeredTransition', {
  stagger: function (index) {
    // 每个过渡项目增加 50ms 延时
    // 但是最大延时限制为 300ms
    return Math.min(300, index * 50)
  }
})
```

第 9 章 Method

Vue.js 的事件监听一般都通过 v-on 指令配置在 HTML 中，虽然也可以在 JavaScript 代码中使用原生 addEventListener 方法添加事件监听，但 Vue.js 本身并不提倡如此。看上去这种方式不符合传统的“关注点分离”（separation of concern）的理念，但其实所有的 Vue.js 事件处理方法和表达式都严格绑定在当前视图的 ViewModel 上。实际上，采用它提供的 v-on 指令有如下几点好处：

- 通过查看 HTML 模板便能轻松定位 JavaScript 代码中对应的方法。
- 无须在 JavaScript 中手动绑定事件，ViewModel 和 DOM 完全解耦，更易于测试。
- 当一个 ViewModel 被销毁时，所有的事件处理器都会自动被删除。

9.1 如何绑定事件

在原生 DOM 事件中，我们可以通过 JavaScript 给 HTML 文档元素注册不同的事件处理程序。代码示例如下：

```
<button onclick="learnVue()">DDFE</button>
```

AngularJS 也采取了类似的方式，只不过换成了 ng-前缀的事件指令：

```
<button ng-click="learnBue()">DDFE</button>
```

类似的，Vue.js 也采取了这样的方式来绑定事件，下面进行详细介绍。

9.1.1 内联方式

Vue.js 在 HTML 文档元素中采用 v-on 指令来监听 DOM 事件，代码示例如下：

```
<div id="example">
```

```
  <button v-on:click="greet">Greet</button>
</div>
```

注：此处语法以 Vue.js 1.0 版本为准，0.12 版本与 1.0 版本的区别见 9.3 节。

这里将一个单击事件处理器 click 绑定到 greet 方法，该方法在 Vue 实例中进行定义。

在这种内联方式下一个事件处理器只能绑定一个方法，如需绑定多个方法，仍需在 JavaScript 代码中使用 addEventListener 方法来绑定。

同样的，类似于原生 JavaScript 以及 AngularJS，除了直接绑定到一个方法外，也可以直接使用内联 JavaScript 语句。代码示例如下：

```
<div id="example-2">
  <button v-on:click="say('hi')">Say Hi</button>
  <button v-on:click="count = count + 1">Say What</button>
</div>
```

与内联表达式相仿，事件处理器限制为一个 JavaScript 语句。

9.1.2　methods 配置

在上一节中，当用户将 click 事件与某个方法绑定时，需要在 Vue 实例当中进行定义，所有定义的方法都放在 methods 属性下。针对上一节的 greet 方法定义代码示例如下：

```
var vm = new Vue({
  el: '#example',
  // 在 `methods` 对象中定义方法
  methods: {
    greet: function (event) {
      // 方法内 `this` 指向 vm
      alert('Welcome to Vue.js By DDFE!')
      // `event` 是原生 DOM 事件
      alert(event.target.tagName)
    }
  }
})

// 也可以在 JavaScript 代码中调用方法
vm.greet()
```

对于 say 方法，可按照如下方式定义：

```
new Vue({
  el: '#example-2',
  methods: {
    say: function (msg) {
      alert(msg)
    }
  }
})
```

需要注意的地方如下：

- methods 中定义的方法内的 this 始终指向创建的 Vue 实例。
- 与事件绑定的方法支持参数 event 即原生 DOM 事件的传入。
- 方法用在普通元素上时，只能监听原生 DOM 事件；用在自定义元素组件上时，也可以监听子组件触发的自定义事件（详见“组件”一章）。

9.1.3　$events 应用

在上一节中，在 Vue 实例中创建的方法需要访问原生 DOM 事件时可以直接传入 event 来获取。如果在内联语句处理器中需要访问原生 DOM 事件时，则可以用一个特殊变量$event 将其传入方法中。代码示例如下：

```
<button v-on:click="say('hello!', $event)">Submit</button>
// ...
methods: {
  say: function (msg, event) {
    // 现在我们可以访问原生事件对象
    event.preventDefault()
  }
}
```

9.2　如何使用修饰符

在第 3 章“指令”中提过，修饰符（modifiers）是以半角句号（.）开始的特殊后缀，用于表示指令应当以特殊方式绑定。在事件处理器上，Vue.js 为 v-on 提供了 4 个事件修饰符，

即.prevent、.stop、.capture 与.self，以使 JavaScript 代码负责处理纯粹的数据逻辑，而不用处理这些 DOM 事件的细节。Vue.js 还为 v-on 添加了按键修饰符，用于监听键盘事件。

在使用方式上，事件修饰符可以串联，代码示例如下：

```
<a v-on:click.stop.prevent="doThat">
```

也可以只有修饰符而不绑定事件，代码示例如下：

```
<form v-on:submit.prevent></form>
```

注：事件修饰符与按键修饰符均为 Vue.js 1.0 版本增加的特性，0.12 版本无法使用。

9.2.1　prevent

在事件处理器中经常需要调用 event.preventDefault()来阻止事件的默认行为，Vue.js 提供了.prevent 事件修饰符以使之在 HTML 中便能完成操作。代码示例如下：

```
<!-- 提交事件不再重载页面 -->
<form v-on:submit.prevent="onSubmit"></form>
```

9.2.2　stop

除了 event.preventDefault()，用于阻止事件冒泡的 event.stopPropagation()也经常被调用，Vue.js 也提供了相应的.stop 事件修饰符。代码示例如下：

```
<!-- 阻止单击事件冒泡 -->
<a v-on:click.stop="doThis"></a>
```

9.2.3　capture

.capture 事件修饰符是 Vue.js 1.0.16 版本中新增的，表示添加事件侦听器时采用 capture 即捕获模式。代码示例如下：

```
<div v-on:click.capture="doThis">...</div>
```

9.2.4　self

.self 事件修饰符同样是 Vue.js 1.0.16 版本中新增的，表示只当事件在该元素本身（而不是子元素）触发时触发回调。代码示例如下：

```
<div v-on:click.self="doThat">...</div>
```

9.2.5　按键

监听键盘事件经常需要检测 keyCode。Vue.js 可以为 v-on 添加键盘修饰符，代码示例如下：

```
<!-- 只有在 keyCode 是 13 时调用 vm.submit()-->
<input v-on:keyup.13="submit">
```

鉴于记住所有的 keyCode 比较困难，Vue.js 为常用的按键提供了别名。代码示例如下：

```
<!-- 同上 -->
<input v-on:keyup.enter="submit">
<!-- 缩写语法，详见下一节 -->
<input @keyup.enter="submit">
```

完整的按键别名如下：

- enter（keycode:13）
- tab（keycode:9）
- delete（keycode:8,46）
- esc（keycode:27）
- space（keycode:32）
- up（keycode:38）
- down（keycode:40）
- left（keycode:37）
- right（keycode:39）

9.3　Vue.js 0.12 到 1.0 中的变化

Vue.js 从 0.12 版本到当前最新的 1.0 版本经历了不小的变化，具体到本章内容，1.0 版本新增的内容包括事件修饰符和按键修饰符。除此之外，还有以下两点语法上的变动。

9.3.1　v-on 变更

在 Vue.js 0.12 版本中，v-on 绑定事件的语法如下：

```
<a v-on="click: myFunction">触发一个方法函数</a>
```

而在 1.0 版本中作了一些改动：

```
<a v-on:click="myFunction">触发一个方法函数</a>
```

有鉴于此，0.12 版本中多重指令从句的形式在 1.0 版本中也不再提供：

```
<!==0.12 版本，多重指令从句==>
<div v-on="
  click   : onClick,
  keyup   : onKeyup,
  keydown : onKeydown
">
</div>

<!==1.0 版本中必须分开绑定==>
<div v-on:click="onClick"v-on:keyup="onKeyup" v-on:keydown="onKeydown">
</div>
```

9.3.2　@click 缩写

Vue.js 在 1.0 版本中为 v-on 提供了缩写形式，在 0.12 版本中则无法使用：

```
<!-- 完整语法 -->
<a v-on:click="doSomething"></a>

<!-- 缩写 -->
<a @click="doSomething"></a>
```

第 10 章

Vue 实例方法

本章我们介绍Vue实例提供的一些有用的属性和方法，这些属性和方法名都以前缀$开头。

10.1 实例属性

在详细讲解每个属性的使用之前，我们先看一下属性总览图，如图 10-1 所示。

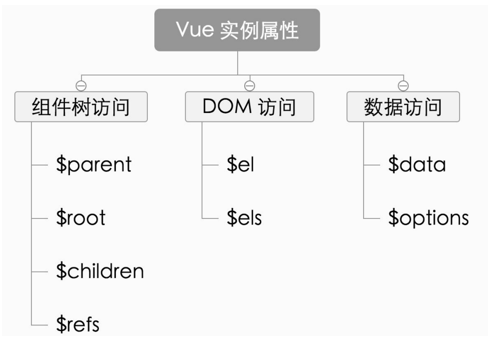


图10-1 Vue实例属性总览

10.1.1 组件树访问

1. $parent

用来访问当前组件实例的父实例。

2. $root

用来访问当前组件树的根实例，如果当前组件没有父实例，$root 表示当前组件实例本身。

3. $children

用来访问当前组件实例的直接子组件实例。

4. $refs

用来访问使用了 v-ref 指令的子组件。v-ref 的详细介绍请参阅 3.1.11 节。

10.1.2　DOM 访问

1. $el

用来访问挂载当前组件实例的 DOM 元素。

2. $els

用来访问$el 元素中使用了 v-el 指令的 DOM 元素。v-el 的详细介绍请参阅 3.1.12 节。

10.1.3　数据访问

1. $data

用来访问组件实例观察的数据对象，该对象引用组件实例化时选项中的 data 属性。

2. $options

用来访问组件实例化时的初始化选项对象。

10.2　实例方法

10.2.1　实例 DOM 方法的使用

在详细讲解每个方法的使用之前，我们先看一下方法总览图，如图 10-2 所示。

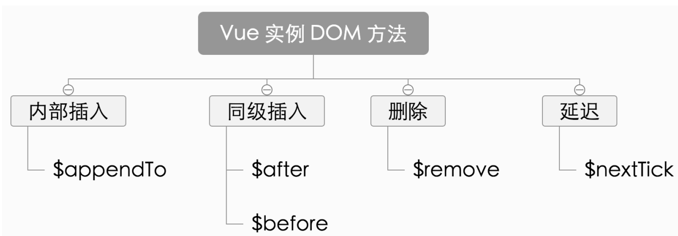


图10-2　Vue实例DOM方法总览

1. $appendTo()

$appendTo()方法用来将 el 所指的 DOM 元素或片段插入到目标元素中。

该方法接受两个参数：

- elementOrSelector（字符串或 DOM 元素），该参数可以是一个选择器字符串或者 DOM 元素。
- callback（可选，函数），回调函数，该回调函数会在 el 元素被插入到目标元素后被触发。注：如果在 el 上应用了过渡效果，则回调会在过渡完成后被触发。

2. $before()

$before()方法用来将 el 所指的 DOM 元素或片段插入到目标元素之前。

该方法接受两个参数：

- elementOrSelector（字符串或 DOM 元素），该参数可以是一个选择器字符串或者 DOM 元素。
- callback（可选，函数），回调函数，该回调函数会在 el 元素被插入到目标元素后被触发。注：如果在 el 上应用了过渡效果，则回调会在过渡完成后被触发。

3. $after()

$after()方法用来将 el 所指的 DOM 元素或片段插入到目标元素之后。

该方法接受两个参数：

- elementOrSelector（字符串或 DOM 元素），该参数可以是一个选择器字符串或者 DOM 元素。
- callback（可选，函数），回调函数，该回调函数会在 el 元素被插入到目标元素后被触发。注：如果在 el 上应用了过渡效果，则回调会在过渡完成后被触发。

4. $remove()

$remove()方法用来将 el 所指的 DOM 元素或片段从 DOM 中删除。

该方法接受一个参数：

- callback（可选，函数），回调函数，该回调函数会在 el 元素在 DOM 中被删除后触发。注：如果在 el 上应用了过渡效果，则回调会在过渡完成后被触发。

5. $nextTick()

$nextTick()方法用来在下次 DOM 更新循环后执行指定的回调函数，使用该方法可以保证 DOM 中的内容已经与最新数据保持同步。

该方法接受一个参数：

- callback（可选，函数），回调函数，该回调函数会在下次 DOM 更新循环后被执行。它和全局的 Vue.nextTick 方法一样，不同的是，callback 中的 this 会自动绑定到调用它的 Vue 实例上。

10.2.2 实例 Event 方法的使用

在详细讲解每个方法的使用之前，我们先看一下 Vue 实例事件总览图，如图 10-3 所示。

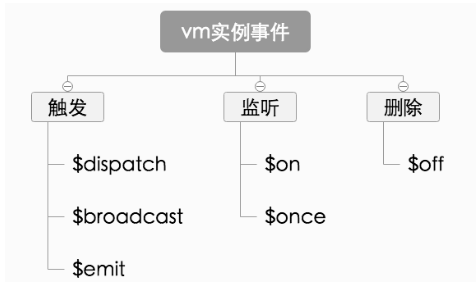


图10-3 Vue实例事件总览

1. $on()

$on()方法用来监听实例上的自定义事件。

该方法接受两个参数：

- event（字符串），该参数可以是一个事件名称。
- callback（函数），回调函数，该回调函数会在执行$emit、$broadcast 或者$dispatch 后触发。

2. $once()

$once()方法也是用来监听实例上的自定义事件，但只触发一次。

该方法接受两个参数：

- event（字符串），该参数可以是一个事件名称。
- callback（函数），回调函数，该回调函数会在执行$emit、$broadcast 或者$dispatch 后触发。

3. $emit()

$emit()方法用来触发事件。

该方法接受两个参数：

- event（字符串），该参数可以是一个事件名称。
- args（可选），传递给监听函数的参数。

4. $dispatch()

$dispatch()方法用来派发事件，即先在当前实例触发，再沿着父链一层一层向上，如果对应的监听函数返回 false 就停止。

该方法接受两个参数：

- event（字符串），该参数可以是一个事件名称。
- args（可选），传递给监听函数的参数。

5. $broadcast()

$broadcase()方法用来广播事件，即遍历当前实例的$children，如果对应的监听函数返回 false 就停止。

该方法接受两个参数：

- event（字符串），该参数可以是一个事件名称。
- args（可选），传递给监听器的参数。

6. $off()

$off()方法用来删除事件监听器。

该方法接受两个参数：

- event（字符串），该参数可以是一个事件名称。
- callback（可选，函数），对应的回调函数。

如果没有参数，即删除所有的事件监听器；如果只提供一个参数——事件名称，即删除它对应的所有监听器；如果提供两个参数——事件名称和回调函数，即删除对应的这个回调函数。

第 11 章 组件

组件是 Vue.js 最推崇的，也是最强大的功能之一，核心目标是为了可重用性高，减少重复性的开发。我们可以把组件代码按照 template、style、script 的拆分方式，放置到对应的.vue 文件中。

Vue.js 的组件可以理解为预先定义好行为的 ViewModel 类。一个组件可以预定义很多选项，但最核心的是以下几个：

- 模板（template）—— 模板声明了数据和最终展现给用户的 DOM 之间的映射关系。
- 初始数据（data）—— 一个组件的初始数据状态。对于可复用的组件来说，通常是私有的状态。
- 接受的外部参数（props）—— 组件之间通过参数来进行数据的传递和共享。参数默认是单向绑定（由上至下），但也可以显式声明为双向绑定。
- 方法（methods）—— 对数据的改动操作一般都在组件的方法内进行。可以通过 v-on 指令将用户输入事件和组件方法进行绑定。
- 生命周期钩子函数（lifecycle hooks）—— 一个组件会触发多个生命周期钩子函数，比如 created、attached、destroyed 等。在这些钩子函数中，我们可以封装一些自定义的逻辑。和传统的 MVC 相比，这可以理解为 Controller 的逻辑被分散到了这些钩子函数中。

11.1　基础

11.1.1　注册

1. 全局注册

```
Vue.component('didi-component', DIDIComponent)
```

如上所示，第一个参数是注册组件的名称（即在 HTML 中我们可以这样使用组件：<didi-component></didi-component>）；第二个参数是组件的构造函数，它可以是 Function，也可以是 Object。

- Function —— DIDIComponent 可以是用 Vue.extend()创建的一个组件构造器。代码示例如下：

```
var MyComponent = Vue.extend({
  // 选项...
})
```

- Object —— DIDIComponent 传入选项对象，Vue.js 在背后自动调用 Vue.extend()。代码示例如下：

```
// 在一个步骤中扩展与注册
Vue.component('didi-component', {
  template: '<div>A custom component!</div>'
})
```

组件在注册之后，便可以在父实例的模块中以自定义元素<didi-component>的形式使用。要确保在初始化根实例之前注册了组件，代码示例如下：

```
<body>
  <div id="example">
    <didi-component></didi-component>
  </div>
</body>
<script>
var DIDIComponent = Vue.extend({
 template: '<div>A custom component!</div>'
})
// 注册
Vue.component('didi-component', DIDIComponent)
 // 创建根实例
```

```
  new Vue({
 el: '#example'
})
</script>
```

效果如图 11-1 所示。

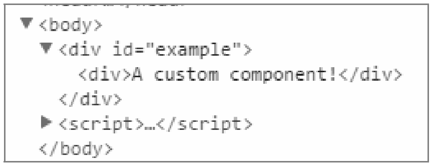


图11-1　全局注册

注：组件的模板替换了自定义元素，自定义元素的作用只是作为一个挂载点。可以用实例选项 replace 决定是否替换自定义元素。

2. 局部注册

不需要每个组件都全局注册，可以让组件只能用在其他组件内。我们可以用实例选项 components 注册，代码示例如下：

```
<body>
  <div id="example">
    <didi-component></didi-component>
  </div>
</body>
<script>
  var Child = Vue.extend({
    template: '<div>i am child!</div>',
    replace: true
  })
  var Parent = Vue.extend({
    template: '<p>i am parent</p><br/><child></child>',
    components: {
      // <didi-component>只能用在父组件模板内
      'child': Child
    }
  })
  // 创建根实例
  new Vue({
    el: '#example',
    components: {
      'didi-component': Parent
    }
  })
</script>
```

效果如图 11-2 所示。

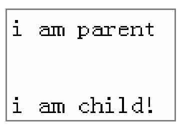


图11-2　局部注册

为了让事件更简单，我们可以直接传入选项对象而不是构造器给 Vue.component()和 components 选项。代码示例如下：

```
// 在一个步骤中扩展与注册
Vue.component('didi-component', {
  template: '<div>A custom component!</div>'
})

// 局部注册也可以这么做
var Parent = Vue.extend({
  components: {
    'didi-component': {
      template: '<div>A custom component!</div>'
    }
  }
})
```

11.1.2　数据传递

总结下来，Vue.js 组件之间有三种数据传递方式：

- props
- 组件通信
- slot

下面我们分别对每一种数据传递方式做详细的阐述。

1. props

“props”是组件数据的一个字段，期望从父组件传下来数据。因为组件实例的作用域是孤立的，这意味着不能并且不应该在子组件的模板内直接引用父组件的数据，所以子组件需要显式地用 props 选项来获取父组件的数据。props 选项可以是字面量，也可以是表达式，还可以绑定修饰符。下面我们详细看一下它是如何使用的。

（1）字面量语法

```
Vue.component('child', {
  // 声明 props
  props: ['msg'],
```

```
  // prop 可以用在模板内
  // 可以用 `this.msg` 设置
  template: '<span>{{ msg }},DDFE!</span>'
})
```



图11-3 字面量

向它传入一个普通字符串，效果如图 11-3 所示。

```
<child msg="hello"></child>
```

HTML 特性不区分大小写。名字形式为 camelCase 的 props 用作特性时，需要转换为 kebab-case 形式（短横线隔开）。代码示例如下：

```
Vue.extend({
  // 声明 props
  props: ['myComponent'],
  template: '<div> {{myComponent}} DDFE! </div>',
  replace: true
})
<!-- kebab-case in HTML -->
<child my-message="hello!"></child>
```

（2）动态语法

类似于用 v-bind 将 HTML 特性绑定到一个表达式，我们也可以用 v-bind 将动态 props 绑定到父组件的数据。每当父组件的数据变化时，该变化也会传导给子组件（类似于 AngularJS 绑定策略中的@）。代码示例如下：

```
var Child = Vue.extend({
  // 声明 props
  props: ['didiProps'],
  template: '<div> {{didiProps }} DDFE! </div>',
  replace: true
})
var Parent = Vue.extend({
  template: '<p>i am parent</p><br/><child :didi-props="hello"></child>',
  data:function () {
    return { 'hello': 'hello,'}
  },
  components: {
    // <child> 只能用在父组件模板内
    'child': Child
  }
})
```

```
// 创建根实例
new Vue({
  el: '#example',
  components: {
    'didi-props': Parent
  }
})
```

（3）绑定修饰符

props 默认是单向绑定——当父组件的属性变化时，将传导给子组件，但是反过来不会。这是为了防止子组件无意修改父组件的状态——这会让应用的数据流难以理解。不过，也可以使用绑定修饰符：

- .sync，双向绑定。
- .once，单次绑定。

代码示例如下：

```
<!-- 默认为单向绑定 -->
<child :msg="parentMsg"></child>
<!-- 双向绑定 -->
<child :msg.sync="parentMsg"></child>
<!-- 单次绑定 -->
<child :msg.once="parentMsg"></child>
```

双向绑定会把子组件的 msg 属性同步回父组件的 parentMsg 属性（类似于 AngularJS 绑定策略中的=）；单次绑定在建立之后不会同步之后的变化。如果 props 是一个对象或数组，那么它是按引用传递的。在子组件内修改它会影响父组件的状态，而不管使用哪种绑定类型。代码示例如下：

```
<body id="example">
  <input type="text" v-model="info.name"/>
  <child v-bind:msg.once="info"></child>
</body>
<script>
    // 创建根实例
    new Vue({
        el: '#example',
        data: function(){
```

```
            return {
                info: {
                    name:'顺风车'
                }
            }
        },
        components: {
            'child': {
                // 声明 props
                props: ['msg'],
                template: '<div>{{msg.name}} DDFamily! </div>'
            }
        }
    })
</script>
```



图11-4 单次绑定

效果如图 11-4 所示。

（4）prop 验证

组件可以为 props 指定验证要求。当组件给其他人使用时，可以确保其他人正确地使用组件。此时 prop 的值是一个对象，代码示例如下：

```
Vue.component('example', {
  props: {
    // 基础类型检测 （`null` 的意思是任何类型都可以）
    propA: 'null',
    // 多种类型 (1.0.21+)
    propM: [String, Number],
    // 必需且是字符串
    propB: {
      type: String,
      required: true
    },
    // 数字，有默认值
    propC: {
      type: Number,
      default: 100
    },
    // 对象/数组的默认值应当由一个函数返回
    propD: {
```

```
      type: Object,
      default: function () {
        return { msg: 'hello' }
      }
    },
    // 指定这个 prop 为双向绑定
    // 如果绑定类型不对将抛出一条警告
    propE: {
      twoWay: true
    },
    // 自定义验证函数
    propF: {
      validator: function (value) {
        return value > 10
      }
    }
  }
})
```

type 可以是下面的原生构造器：

- String
- Number
- Boolean
- Object
- Function
- Array

type 也可以是一个自定义构造器，使用 instanceof 检测。

当 props 验证失败时，Vue.js 将拒绝在子组件上设置此值，如果使用的是开发版本，将会抛出一条警告。

（5）prop 转换函数

现在我们可以给 prop 定义一个 coerce 函数（Vue.js 1.0.12 新增）——每当 prop 在父类更新的时候，prop 的值将会通过 coerce 函数，可以将它理解为 prop 的单向过滤器，只是它被定义在

了子类组件内。

```
Vue.component('example', {
  props: {
    // 转换函数（1.0.12 新增）
    // 在设置值之前转换值
    propG: {
      coerce: function (val) {
        return val + '' // 将值转换为字符串
      }
    },
    propH: {
      coerce: function (val) {
        return JSON.parse(val) // 将 JSON 字符串转换为对象
      }
    }
  }
)
```

（6）props & propsData & data

以上属性均与数据有关，props 上面已经介绍，它作用于父子组件之间的数据传递；而 data 我们在开发组件或实例化 Vue 对象时经常使用，它作为组件的私有数据存在；propsData 这个属性则常用来在组件初始化后覆盖 props 中的属性。

2. 组件通信

类似于 AngularJS，Vue.js 也实现了组件之间的相互通信。尽管子组件可以用 this.$parent 访问它的父组件，父组件有一个数组 this.$children，包含它所有的子元素，根实例的后代可以用 this.$root 访问根实例，不过子组件应当避免直接依赖父组件的数据，尽量显式地使用 props 传递数据。另外，在子组件中修改父组件的状态是非常糟糕的做法，因为：

- 父组件与子组件紧密地耦合。
- 只看父组件，很难理解父组件的状态，因为它可能被任意子组件修改！在理想情况下，只有组件自己能修改其状态。

因为作用域是有层次的，所以我们可以在作用域链上传递事件。通常来说，选择事件传递方式，一个好的经验规则就是：查看要触发事件的作用域。如果要通知整个事件系统，就要向

下广播。每个 Vue 实例都是一个事件触发器：

- $on()——监听事件。
- $emit()——把事件沿着作用域链向上派送。
- $dispatch()——派发事件，事件沿着父链冒泡。
- $broadcast()——广播事件，事件向下传导给所有的后代。

在解决组件之间通信问题前，我们先来看一下 Vue 的自定义事件接口，用于在组件树中通信。这个事件系统独立于原生 DOM 事件，用法也不同，代码示例如下：

```
<body>
  <!-- 子组件模板 -->
  <template id="child-template">
    <input v-model="msg">
    <button v-on:click="notify">Dispatch Event</button>
  </template>
  <!-- 父组件模板 -->
  <div id="events-example">
    <p>Messages: {{ messages | json }}</p>
    <child></child>
  </div>
</body>
<script>
  // 注册子组件
  // 将当前消息派发出去
  Vue.component('child', {
    template: '#child-template',
    data: function () {
      return { msg: 'hello' }
    },
    methods: {
      notify: function () {
        if (this.msg.trim()) {
          this.$dispatch('child-msg', this.msg)
          this.msg = ''
        }
      }
    }
  })
```

```
  // 初始化父组件
  // 收到消息时将事件推入一个数组中
  var parent = new Vue({
    el: '#events-example',
    data: {
      messages: []
    },
    // 在创建实例时 `events` 选项简单地调用 `$on`
    events: {
      'child-msg': function (msg) {
        // 事件回调内的 `this` 自动绑定到注册它的实例上
        this.messages.push(msg)
      }
    }
  })
</script>
```

效果如图 11-5 所示。

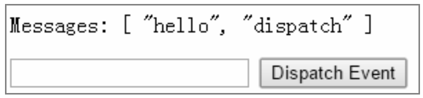


图11-5　通信

上面展示了父子组件的通信，只是从父组件的代码中不能直观地看到 child-msg 事件来自哪里。如果在模板中子组件用到的地方声明事件处理器则会更好。为此，子组件可以用 v-on 监听自定义事件。代码示例如下：

```
<body>
  <!-- 子组件模板 -->
  <template id="child-template">
    <input v-model="msg">
    <button v-on:click="notify">Dispatch Event</button>
  </template>
  <!-- 父组件模板 -->
  <div id="events-example">
    <p>Messages: {{ messages | json }}</p>
    <child v-on:child-msg="handleIt"></child>
  </div>
</body>
<script>
  // 注册子组件
  // 将当前消息派发出去
  Vue.component('child', {
    template: '#child-template',
```

```
    data: function () {
      return { msg: 'hello' }
    },
    methods: {
      notify: function () {
        if (this.msg.trim()) {
          this.$dispatch('child-msg', this.msg)
          this.msg = ''
        }
      }
    }
  })

  // 初始化父组件
  // 在收到消息时将事件推入一个数组中
  var parent = new Vue({
    el: '#events-example',
    data: {
      messages: []
    },
    methods:{
      'handleIt': function(){
        alert("a")
      }
    },
    // 在创建实例时 `events` 选项简单地调用 `$on`
    events: {
      'child-msg': function (msg) {
        // 事件回调内的 `this` 自动绑定到注册它的实例上
        this.messages.push(msg)
      }
    }
  })
</script>
```

这样就很清楚了——当子组件触发了 child-msg 事件时，父组件的 handleIt 方法将被调用。所有影响父组件状态的代码都放到父组件的 handleIt 方法中；子组件只关注触发事件。

尽管有 props 和 events，但是有时仍然需要在 JavaScript 中直接访问子组件。为此，可以使

用 v-ref 为子组件指定一个索引 ID。代码示例如下：

```
<comp v-ref:child></comp>
<comp v-ref:some-child></comp>
// 从父组件访问
this.$refs.child
this.$refs.someChild
```

3. slot 分发内容

在使用组件时，常常要像这样组合它们：

```
<didi>
  <didi-header></didi-header>
  <didi-footer></didi-footer>
</didi>
```

注意两点：

- <didi>组件不知道它的挂载点会有什么内容，挂载点的内容是由<didi>的父组件决定的。
- <didi>组件很可能有它自己的模板。

为了让组件可以组合，我们需要一种方式来混合父组件的内容与子组件自己的模板。这个处理称为内容分发（或“transclusion”，如果熟悉 AngularJS 的话）。Vue.js 实现了一个内容分发 API，参照了当前 Web 组件规范草稿，使用特殊的<slot>元素作为原始内容的插槽。

（1）编译作用域

在深入内容分发 API 之前，我们先明确内容的编译作用域。假定模板为：

```
<child v-on:child-msg="handleIt">{{msg}}</child>
```

msg 应该绑定到父组件的数据，还是绑定到子组件的数据？答案是父组件。简单地说，组件作用域是：父组件模板的内容在父组件作用域内编译；子组件模板的内容在子组件作用域内编译。

一个常见的错误是试图在父组件模板内将一个指令绑定到子组件的属性/方法：

```
<!-- 无效 -->
<child-component v-show="someChildProperty"></child-component>
```

假定 someChildProperty 是子组件的属性，上例不会如预期的那样工作。父组件模板不应该知道子组件的状态。

如果要将子组件内的指令绑定到一个组件的根节点，则应当在它的模板内这样做：

```
Vue.component('child-component', {
  // 有效，因为是在正确的作用域内
  template: '<div v-show="someChildProperty">Child</div>',
  data: function () {
    return {
      someChildProperty: true
    }
  }
})
```

类似的，分发内容是在父组件作用域内编译的。

（2）单个 slot

父组件的内容将被抛弃，除非子组件模板包含<slot>。如果子组件模板只有一个没有特性的 slot，父组件的整个内容将插到 slot 所在的地方并替换它。

<slot>标签的内容视为回退内容。回退内容在子组件的作用域内编译，当宿主元素为空并且没有内容供插入时显示这个回退内容。

假定 didi-component 组件有下面模板：

```
<div>
  <h1>This is my component!</h1>
  <slot>
    如果没有分发内容则显示我。
  </slot>
</div>
```

父组件模板：

```
<didi-component>
  <p>This is some original content</p>
  <p>This is some more original content</p>
</didi-component>
```

渲染结果为：

```
<div>
  <h1>This is my component!</h1>
  <p>This is some original content</p>
  <p>This is some more original content</p>
</div>
```

（3）具名 slot

<slot> 元素可以用一个特殊特性 name 配置如何分发内容。多个 slot 可以有不同的名字。具名 slot 将匹配内容片段中有对应 slot 特性的元素。

仍然可以有一个匿名 slot，作为找不到匹配的内容片段的回退插槽，它是默认 slot。如果没有默认 slot，这些找不到匹配的内容片段将被抛弃。

例如，假定有一个 multi-insertion 组件，代码示例如下：

```
<div>
  <slot name="one"></slot>
  <slot></slot>
  <slot name="two"></slot>
</div>
```

父组件模板：

```
<multi-insertion>
  <p slot="one">One</p>
  <p slot="two">Two</p>
  <p>Default A</p>
</multi-insertion>
```

渲染结果为：

```
<div>
  <p slot="one">One</p>
  <p>Default A</p>
  <p slot="two">Two</p>
</div>
```

在组合组件时，内容分发 API 是非常有用的机制。

（4）原理解析

slot 是一个内置的自定义元素指令，源码定义如下：

```
{
  priority: SLOT,
  params: ['name'],
  bind () {
    var name = this.params.name || 'default'
    /*this.vm._slotContents 为父模板中子组件的模板内容按照 slot 的值组成的键值对（其中没有 slot
```

```
  属性的将被放入'default'值中）*/
  var content = this.vm._slotContents && this.vm._slotContents[name]
  if (!content || !content.hasChildNodes()) {
    this.fallback()
  } else {
    this.compile(content.cloneNode(true), this.vm._context, this.vm)
  }
},
compile (content, context, host) {
  if (content && context) {
    if (
      this.el.hasChildNodes() &&
      content.childNodes.length === 1 &&
      content.childNodes[0].nodeType === 1 &&
      content.childNodes[0].hasAttribute('v-if')
    ) {
      /*如果父模板有类似数据<p slot="one" v-if="show">子模板<slot name="one">hello</slot>
      content 内容追加 template 标签带 v-else 属性文本为 hello*/
      const elseBlock = document.createElement('template')
      elseBlock.setAttribute('v-else', '')
      elseBlock.innerHTML = this.el.innerHTML
      elseBlock._context = this.vm
      content.appendChild(elseBlock)
    }
    const scope = host
      ? host._scope
      : this._scope
    this.unlink = context.$compile(
      content, host, scope, this._frag
    )
  }
  if (content) {
    replace(this.el, content)
  } else {
    remove(this.el)
  }
},
fallback () {
  this.compile(extractContent(this.el, true), this.vm)
```

```
  },
  unbind () {
    if (this.unlink) {
      this.unlink()
    }
  }
}
```

slot 在 bind 回调函数中，根据 name 获取将要替换插槽的元素，如果在上下文环境中有所需替换的内容，则调用父元素的 replaceChild 方法，用替换元素将 slot 元素替换；否则直接删除将要替换的元素。如果替换插槽元素中有一个顶级元素，且顶级元素的第一子节点为 DOM 元素，且该节点有 v-if 指令，且 slot 元素中有内容，则替换模板将增加 v-else 模板放入插槽中的内容，如果 v-if 指令为 false，则渲染 else 模板内容。

11.1.3　混合

混合以一种灵活的方式为组件提供分布复用功能。混合对象可以包含任意的组件选项。当组件使用了混合对象时，混合对象的所有选项将被“混入”组件自己的选项中。代码示例如下：

```
var myMixin = {
  created: function () {
    this.hello()
  },
  methods: {
    hello: function () {
      console.log('hello from mixin!')
    }
  }
}
// 定义一个组件，使用这个混合对象
var component = Vue.extend({
  mixins: [myMixin],
    template:"<h1>hello~ DIDI</h1>"
  })
  // 创建根实例
  new Vue({
    el: '#example',
    components: {
```

```
    'my-component': component
   }
  })
})
```



图11-6　混合

效果如图 11-6 所示。

当混合对象与组件包含同名选项时，这些选项将以适当的策略合并。例如，同名钩子函数被并入一个数组中，因而都会被调用。另外，混合的钩子将在组件自己的钩子之前调用，代码示例如下：

```
var myMixin = {
  created: function () {
    this.hello()
  },
  methods: {
    hello: function () {
      console.log('hello from mixin!')
    }
  }
}
// 定义一个组件，使用这个混合对象
var component = Vue.extend({
  mixins: [myMixin],
  template:"<h1>hello~ DIDI</h1>",
  created: function () {
    console.log('component hook called')
  }
})
// 创建根实例
new Vue({
  el: '#example',
  components: {
    'my-component': component
  }
}
```

值为对象的选项，如 methods、components 和 directives 将合并到同一个对象内。如果键冲突，则组件的选项优先。代码示例如下：

```
var myMixin = {
     methods: {
```

```
        foo: function () {
          console.log('foo')
        },
        conflicting: function () {
          console.log('from mixin')
        }
    }
}
// 定义一个组件，使用这个混合对象
var myMixin = {
  methods: {
    foo: function () {
      console.log('foo')
    },
    conflicting: function () {
      console.log('from mixin')
    }
  }
}
// 定义一个组件，使用这个混合对象
var component = Vue.extend({
  mixins: [myMixin],
  template:'<h1>hello~ DIDI</h1>',
  methods: {
    bar: function () {
      console.log('bar')
    },
    conflicting: function () {
      console.log('from self')
    }
  }
})
var vm = new component();
vm.foo() // -> "foo"
vm.bar() // -> "bar"
vm.conflicting() // -> "from self"
```

注：Vue.extend()使用同样的合并策略。

混合也可以全局注册，但需要小心使用！一旦全局注册混合，它就会影响所有之后创建的

Vue 实例。如果使用恰当，则可以为自定义选项注入处理逻辑。代码示例如下：

```
Vue.mixin({
  created: function () {
  var myOption =  this.$options.myOption
      if (myOption) {
        console.log(myOption)
      }
  }
})
// 定义一个组件，使用这个混合对象
var component = Vue.extend({
  template:"<h1>hello~ DIDI</h1>"
})
new Vue({
  el:'#example',
  components: {
      'my-component': component
  },
  myOption:'hello'
})
```

慎用全局混合，因为它会影响到每个所创建的 Vue 实例，包括第三方组件。在大多数情况下，它应当只用于自定义选项，就像上面示例一样。

11.1.4　动态组件

多个组件可以使用同一个挂载点，然后动态地在它们之间切换。使用保留的<component>元素，动态地绑定到它的 is 特性。代码示例如下：

```
<body id="example">
  <input type="radio" id="one" value="fast" v-model="currentView">
  <label for="one">fast</label>
  <br>
  <input type="radio" id="two" value="bus" v-model="currentView">
  <label for="two">bus</label>
  <br>
  <input type="radio" id="two" value="business" v-model="currentView">
  <label for="two">business</label>
  <template id="bus">
```

```
    <div>滴滴巴士</div>
  </template>
  <template id="business">
    <div>滴滴专车</div>
  </template>
  <template id="fast">
    <div>滴滴快车</div>
  </template>
  <component :is="currentView">
    <!-- 组件在 vm.currentview 变化时改变 -->
  </component>
</body>
<script>
  var bus = Vue.extend({
    template: '#bus',
    replace: true
  });
  var business = Vue.extend({
    template: '#business',
    replace: true
  })
  var fast = Vue.extend({
    template: '#fast',
    replace: true
  })
  // 创建根实例
  new Vue({
    el: '#example',
    data: {
      currentView: 'fast'
    },
    components: {
      fast: fast,
      bus: bus,
      business: business
    }
  })
</script>
```

通过 is 属性绑定的 vm.currentView 变量值，控制展示的组件，效果如图 11-7 所示。

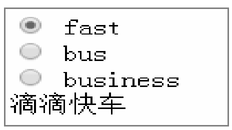


图11-7　动态组件

1. keep-alive

如果把切换出去的组件保留在内存中，则可以保留它的状态或避免重新渲染。为此，可以添加一个 keep-alive 指令参数。代码示例如下：

```
<component :is="currentView" keep-alive>
  <!-- 非活动组件将被缓存 -->
</component>
```

源码定义如下：

```
<!--源码目录：src/util/env.js 7 行-->
bind: function () {
…… // 省略
    // keep-alive cache
    this.keepAlive = this.params.keepAlive
    if (this.keepAlive) {
      this.cache = {}
    }
    // 省略
},

build: function (extraOptions) {
  var cached = this.getCached()
  if (cached) {
    return cached
  }
}
```

Vue.js 为其组件设计了一个[keep-alive]的特性，如果这个特性存在，那么在组件被重复创建时，会通过缓存机制快速创建组件，以提升视图更新的性能。

2. activate 钩子

在切换组件时，切入组件在切入前可能需要进行一些异步操作。为了控制组件切换时长，给切入组件添加 activate 钩子函数。代码示例如下：

```
Vue.component('activate-example', {
  activate: function (done) {
    var self = this
    loadDataAsync(function (data) {
```

```
      self.someData = data
      done()
    })
  }
})
```

注：activate 钩子只作用于动态组件切换或静态组件初始化渲染的过程中，不作用于使用实例方法手工插入的过程中。

3. transition-mode

transition-mode 特性用于指定两个动态组件之间如何过渡。

在默认情况下，进入与离开平滑地过渡。这个特性可以指定另外两种模式：

- in-out —— 新组件先过渡进入，等它的过渡完成之后当前组件过渡出去。
- out-in —— 当前组件先过渡出去，等它的过渡完成之后新组件过渡进入。

```
<!-- 先淡出再淡入 -->
<component
  :is="view"
  transition="fade"
  transition-mode="out-in">
</component>

.fade-transition {
  transition: opacity .3s ease;
}
.fade-enter, .fade-leave {
  opacity: 0;
}
```

11.2　相关拓展

11.2.1　组件和 v-for

自定义组件可以像普通元素一样直接使用 v-for，代码示例如下：

```
<didi-component v-for="item in items"></didi-component>
```

因为组件的作用域是孤立的，上面的代码无法将数据传递到组件内部。在 11.1.2 节中我们

讲述了组件之间通信的三种方式，这里将使用 props。代码示例如下：

```
<didi-componentv-for="item in items":item="item" :index="$index"></didi-component>
```

注：显式声明数据来自哪里可以让组件复用在其他地方。

11.2.2　编写可复用组件

在编写组件时，时刻考虑组件是否可复用是有好处的。一次性组件跟其他组件紧密耦合没关系，但是可复用组件一定要定义一个清晰的公开接口。

Vue.js 组件 API 来自三部分——prop、事件和 slot，关于这三部分上面已作详尽说明。

- prop 允许外部环境传递数据给组件。
- 事件允许组件触发外部环境的 action。
- slot 允许外部环境将内容插入到组件的视图结构内。

使用 v-bind 和 v-on 的简写语法，模板的缩进清楚且简洁。代码示例如下：

```
<my-component
  :foo="baz"
  :bar="qux"
  @event-a="doThis"
  @event-b="doThat">
  <!-- content -->
  <img slot="icon" src="...">
  <p slot="main-text">Hello!</p>
</my-component>
```

11.2.3　异步组件

在大型应用中，我们可能需要将应用拆分为小块，每小块实现按需加载。为了让事情更简单，Vue.js 允许将组件定义为一个工厂函数，动态地解析组件的定义。Vue.js 只在组件需要渲染时触发工厂函数，并且把结果缓存起来，方便后面的再次渲染。代码示例如下：

```
Vue.component('async-example', function (resolve, reject) {
  setTimeout(function () {
    resolve({
      template: '<div>I am async!</div>'
    })
  }, 1000)
})
```

工厂函数接受一个 resolve 回调，在收到从服务器下载的组件定义时调用。也可以调用 reject(reason)指示加载失败。这里 setTimeout 只是为了演示。怎么获取组件完全由我们决定，推荐配合使用 Webpack 的代码分割功能。代码示例如下：

```
Vue.component('async-webpack-example', function (resolve) {
  // 这个特殊的 require 语法告诉 Webpack，将编译后的代码分割成不同的块
  // 这些块将通过 AJAX 请求自动下载
  require(['./my-async-component'], resolve)
})
```

11.2.4 资源命名约定

一些资源如组件和指令，是以 HTML 特性或 HTML 自定义元素的形式出现在模板中的。因为 HTML 特性的名字和标签的名字不区分大小写，所以资源的名字通常需使用 kebab-case 而不是 camelCase 形式，这不大方便。

Vue.js 支持资源的名字使用 camelCase 或 PascalCase 形式，并且在模板中自动将它们转换为 kebab-case 形式（类似于 prop 的命名约定）。代码示例如下：

```
// 在组件定义中
components: {
  // 使用 camelCase 形式注册
  myComponent: { *... */ }
}
<!-- 在模板中使用 kebab-case 形式 -->
<my-component></my-component>
```

ES 6 对象字面量缩写也没问题：

```
// PascalCase
import TextBox from './components/text-box';
import DropdownMenu from './components/dropdown-menu';

export default {
  components: {
    // 在模板中写作 <text-box> 和 <dropdown-menu>
    TextBox,
    DropdownMenu
  }
}
```

11.2.5　内联模板

如果子组件有 inline-template 特性，组件将把它的内容当作其模板，而不是把它当作分发内容且 inline-template 优先级比 template 高。这让模板更灵活。代码示例如下：

```
<didi-component inline-template>
  <p>These are compiled as the component's own template</p>
  <p>Not parent's transclusion content.</p>
</didi-component>
```

但是 inline-template 让模板的作用域难以理解，并且不能缓存模板编译结果。最佳实践是使用 template 选项在组件内定义模板。代码示例如下：

```
<body id="example">
  <div id="items">
    <b>inline-template:</b>
      <my-item  v-for="item in items" :item="item" inline-template>
        <p>{{a}} , {{item.text}}</p>
      </my-item>
      <b>no-inline-template:</b>
      <my-item  v-for="item in items" :item="item">
        {{a}}  , {{items[0].text}}
    </my-item>
  </div>
</body>
<script>
 var items = [
    { number:1, text:'one' },
    { number:2, text:'two' }
  ];
  var vue = new Vue({
    el: '#items',
    data: function(){
      return {
        items: items ,
        a:"i am in parent"
      }
    },
    components: {
      'my-item': {
```

```
        props: ['item'],
        data: function(){
          return {
              a: 'i am in child'
          }
        },
        template:'<p>item 模板：{{a}},{{item.text}}; slot: <slot></slot></p>'
      }
    }
  });
</script>
```

效果如图 11-8 所示。

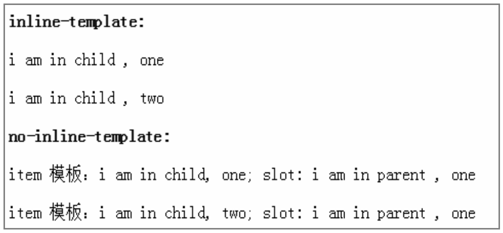


图11-8　内联模板

11.2.6　片段实例

在使用 template 选项时，模板的内容将替换实例的挂载元素，因而推荐模板的顶级元素始终是单个元素。

不要这样写模板：

```
<div>root node 1</div>
<div>root node 2</div>
```

推荐这样写：

```
<div>
  I have a single root node!
  <div>node 1</div>
  <div>node 2</div>
</div>
```

下面几种情况会让实例变成一个片段实例：

- 模板包含多个顶级元素。
- 模板只包含普通文本。
- 模板只包含其他组件（其他组件可能是一个片段实例）。
- 模板只包含一个元素指令，如<partial>或 vue-router 的<router-view>。
- 模板根节点有一个流程控制指令，如 v-if 或 v-for。

这些情况让实例有未知数量的顶级元素，它将把其 DOM 内容当作片段。片段实例仍然会正确地渲染内容。不过，它没有一个根节点，它的$el 指向一个锚节点，即一个空的文本节点（在开发模式下是一个注释节点）。

但是更重要的是，组件元素上的非流程控制指令，非 prop 特性和过渡将被忽略，因为没有根元素供绑定。代码示例如下：

```
<!-- 不可以，因为没有根元素 -->
<example v-show="ok" transition="fade"></example>
<!-- props 可以 -->
<example :prop="someData"></example>
<!-- 流程控制可以，但是不能有过渡 -->
<example v-if="ok"></example>
```

当然片段实例有它的用处，不过通常给组件一个根节点比较好。它会保证组件元素上的指令和特性能正确地转换，同时性能也稍微好些。

11.3　生命周期

在 Vue.js 中，在实例化 Vue 之前，它们以 HTML 的文本形式保存在文本编辑器中。当实例化后将经历创建、编译和销毁三个主要阶段，如图 11-9 所示。

生命周期钩子：

1. init

在实例开始初始化时同步调用。此时数据观测、事件和 Watcher 都尚未初始化。

2. created

在实例创建之后同步调用。此时实例已经结束解析选项，这意味着已建立：数据绑定、计算属性、方法、Watcher/事件回调。但是还没有开始 DOM 编译，$el 还不存在。

3. beforeCompile

在编译开始前调用。

4. compiled

在编译结束后调用。此时所有的指令已生效，因而数据的变化将触发 DOM 更新。但是不担保 $el 已插入文档。

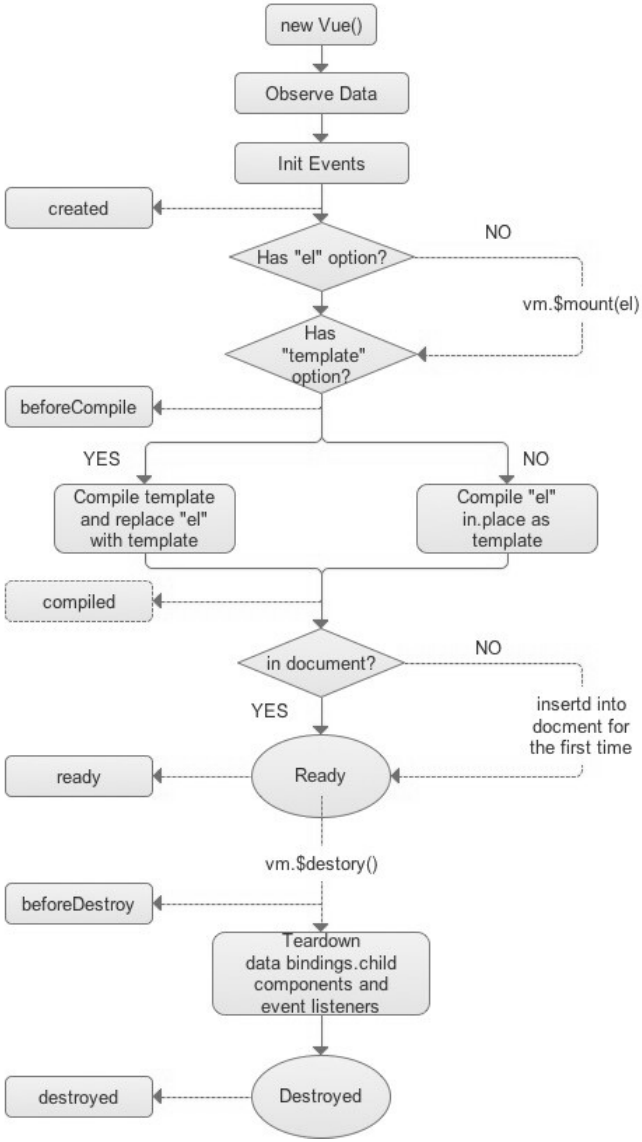


图11-9　生命周期

5. ready

在编译结束和 $el 第一次插入文档之后调用，如在第一次 attached 钩子之后调用。注意，必须是由 Vue 插入（如 vm.$appendTo()等方法或指令更新）才触发 ready 钩子的。

6. attached

vm.$el 插入 DOM 时调用。必须是由指令或实例方法（如$appendTo()）插入，直接操作 vm.$el 不会触发这个钩子。

7. detached

在 vm.$el 从 DOM 中删除时调用。必须是由指令或实例方法删除，直接操作 vm.$el 不会触发这个钩子。

8. beforeDestroy

在开始销毁实例时调用。此时实例仍然有功能。

9. destroyed

在实例被销毁之后调用。此时所有的绑定和实例的指令已经解绑，所有的子实例也已经被销毁。如果有离开过渡，destroyed 钩子在过渡完成之后调用。

11.4　开发组件

11.4.1　基础组件

如果我们要开发更大型的网页或 Web 应用，Web 组件化的思维是非常重要的，这也是今天整个前端社区长久不衰的话题。那么如何开发一个组件呢？

在以往的一些小型的前端项目中，我们习惯把逻辑（script）、视图（view）和样式（style）分开在独立的目录当中，保证三者不耦合在一起。但是随着项目越来越大，这样的结构会让开发越来越痛苦，比如要增加或修改某个 view 时，就要在 script 和 sytle 里找到对应这个 view 的逻辑和样式进行修改。

为了避免随着项目增大带来的难于维护，我们开始尝试前端组件化，把 view 拆分成不同的组件（component），为单个组件编写对应的逻辑和样式：

```
app/components
├── picker
│   ├── picker.ejs
│   ├── picker.js
│   └── picker.css
└── button
    ├── button.ejs
    ├── button.js
    └── button.css
```

这样的开发模式，不仅可以提高代码的可维护性和可重用性，还有利于团队之间的协作，一个组件由一个人去维护，更好地实现分治。幸运的是，随着 React 越来越火，组件化的开发模式也就越来越被大家接受了。

在 Vue.js 中，可以利用一个.vue 文件实现组件化，而不需要对每个组件分别建立 style、script 和 view。这样做的好处是使组件更加直观，而坏处是目前有些编辑器对.vue 的语法支持还是不太好。每个文件就是一个组件，同时还包含了组件之间的依赖关系，整个组件从外观到结构到特性再到依赖关系都一览无余。下面给出一个简单的 button 组件。代码示例如下：

```
<template>
  <button class="didi-btn"
    :class="{
        'disable': disabled,
        'didi-btn-highlight': highlight
    }"
    @click="handleClick($event)"><slot></slot></button>
</template>

<script>
export default {
  name: 'dd-button',
  props: {
    disabled: Boolean,
    highlight: Boolean
  },
  methods: {
    handleClick: function (event) {
      if (this.disabled) {
        event.preventDefault()
        event.stopPropagation()
      }
    }
  }
}
</script>

<style>
```

```
.didi-btn {
   display: block;
   padding: 0;
   width: 100%;
   height: 40px;
   line-height: 37px;
   border-radius: 4px;
   cursor: pointer;
   text-align: center;
   vertical-align: middle;
   touch-action: manipulation;
   background-image: none;
   white-space: nowrap;
   outline: none;
   font-size: 16px;
   color: #878787;
   background: #fafafa;
   border: 1px solid #ccc;
   box-sizing: border-box
}
.didi-btn.active,.didi-btn:active {
   background: #ebebeb
}
.didi-btn.disable,.didi-btn:disabled {
   background: #fafafa;
   border: 1px solid #e5e5e5;
   color: #ccc
}
.didi-btn-highlight {
   background: #fa8919;
   color: #fff;
   border: none
}
.didi-btn-highlight.active,.didi-btn-highlight:active {
   background: #e67e17
}
.didi-btn-highlight.disable,.didi-btn-highlight:disabled {
   background: #e5e5e5;
   color: #fff;
   border: none
}
</style>
```

同时，还可以在*.vue 文件中使用其他预处理器：

```
<style lang="stylus">
  .my-component h2
  color red
</style>

<template lang="jade">
  div.my-component
  h2 Hello from {{msg}}
</template>

<script lang="babel">
// 利用 Babel 编译 ES2015
export default {
  data () {
    return {
      msg: 'Hello from Babel!'
    }
  }
}
```

使用 Webpack 就可以自动将.vue 文件编译成正常的 JavaScript 代码，我们只需要在 Webpack 中配置好 vue-loader 即可（关于 Webpack，第 25 章将详细介绍）。代码示例如下：

```
module.exports = {
  entry: {
    app: './src/main.js'
  },
  output: {
    path: config.build.assetsRoot,
    publicPath: config.build.assetsPublicPath,
    filename: '[name].js'
  },
  resolve: {
    extensions: ['', '.js', '.vue'],
    fallback: [path.join(__dirname, '../node_modules')],
    alias: {
      'src': path.resolve(__dirname, '../src'),
```

```
      'assets': path.resolve(__dirname, '../src/assets'),
      'components': path.resolve(__dirname, '../src/components')
    }
  },
  resolveLoader: {
    fallback: [path.join(__dirname, '../node_modules')]
  },
  module: {
    preLoaders: [
      {
        test: /\.vue$/,
        loader: 'eslint',
        include: projectRoot,
        exclude: /node_modules/
      },
      {
        test: /\.js$/,
        loader: 'eslint',
        include: projectRoot,
        exclude: /node_modules/
      }
    ],
    loaders: [
      {
        test: /\.vue$/,
        loader: 'vue'
      },
      {
        test: /\.js$/,
        loader: 'babel',
        include: projectRoot,
        exclude: /node_modules/
      },
      {
        test: /\.json$/,
        loader: 'json'
      },
      {
        test: /\.html$/,
        loader: 'vue-html'
```

```
      },
      {
        test: /\.(png|jpe?g|gif|svg)(\?.*)?$/,
        loader: 'url',
        query: {
          limit: 10000,
          name: utils.assetsPath('img/[name].[hash:7].[ext]')
        }
      },
      {
        test: /\.(woff2?|eot|ttf|otf)(\?.*)?$/,
        loader: 'url',
        query: {
          limit: 10000,
          name: utils.assetsPath('fonts/[name].[hash:7].[ext]')
        }
      }
    ]
  },
  eslint: {
    formatter: require('eslint-friendly-formatter')
  },
  vue: {
    loaders: utils.cssLoaders()
  }
}
```

像上面的组件格式，把一个组件的模板、样式、逻辑三要素整合在同一个文件中，既方便开发，也方便复用和维护。另外，Vue.js 本身支持对组件的异步加载，配合 Webpack 的分块打包功能，可以极其轻松地实现组件的异步按需加载。

11.4.2 基于第三方组件开发

很多时候我们在开发一些组件时，需要引入第三方组件库，对其进行封装以满足项目需求。下面介绍如何基于 Chart.js 开发。代码示例如下：

```
<template>
  <canvas class="vchart {{chartType}}-chart" v-el:chart-canvas :width="width":height=
"height">
  </canvas>
```

```
</template>
<script>
import Chart from 'chart.js'

module.exports = {
  props: {
    chartType: {
      type: String,
      default: 'line'
    },
    width: {
      type: Number
    },
    height: {
      type: Number
    },
    labels: {
      type: Array,
      validator (value) {
        return value.every(label => typeof label === 'string')
      },
      default () { return [] }
    },
    datasets: {
      type: Array,
      validator (value) {
        return value.every(series => {
          return Array.isArray(series.data) && series.data.every(val => {
            return typeof val === 'number'
          })
        })
      },
      coerce (val) {
        return JSON.parse(JSON.stringify(val))
      },
      default () { return [] }
    },
    options: {
      type: Object,
      default () { return {} }
```

```
    },
    responsive: {
      type: Boolean,
      default: null
    },
    legend: {
      coerce (val) {
        if (typeof val === 'boolean') {
          return {display: val}
        }
      },
      default: null
    }
  },
  methods: {
    parseCommonOptions (options) {
      // responsive
      if (this.responsive !== null) {
        options.responsive = this.responsive
      }

      // legend
      if (this.legend !== null) {
        options.legend = this.legend
      }

      return options
    }
  },
  computed: {
    chartData () {
      return {
        labels: this.labels,
        datasets: this.datasets
      }
    },
    chartOptions () {
      let options = {}
      options = this.parseCommonOptions(options)
```

```
      if (this.parseCustomOptions) {
        options = this.parseCustomOptions(options)
      }

      return Object.assign(this.options, options)
    }
  },
  watch: {
    datasets: {
      // don't need deep
      handler (val, oldVal) {
        this.chartInstance.data.datasets = val
        this.chartInstance.update()
      }
    }
  },
  data () {
    return {
      chartInstance: null
    }
  },
  ready () {
    const chartCanvas = this.$els.chartCanvas
    const ctx = chartCanvas.getContext('2d')
    this.chartInstance = new Chart(ctx, {
      type: this.chartType,
      data: this.chartData,
      options: this.chartOptions
    })
  }
}
</script>
```

组件使用：

```
<template>
  <div id="app">
    <dd-chart :datasets="datasets" :labels="labels" :width="width" :height="height">
    </dd-chart>
  </div>
</template>
```

```
<script>
import ddChart from './components/ddChart

export default {
  data: function () {
    return {
      width: 500,
      height: 200,
      labels: ['1 月', '2 月', '3 月', '4 月', '5 月', '6 月'],
      datasets: [{
        label: '数据一',
        data: [110, 90, 100, 81, 106, 75, 88]
      },
      {
        label: '数据二',
        data: [100, 80, 90, 71, 96, 65, 78]
      },
      {
        label: '数据三',
        data: [90, 70, 80, 61, 86, 55, 68]
      },
      {
        label: '数据四',
        data: [80, 60, 70, 51, 76, 45, 58]
      },
      {
        label: '数据五',
        data: [70, 50, 60, 41, 66, 35, 48]
      },
      {
        label: '数据六',
        data: [60, 40, 50, 31, 56, 25, 38]
      },
      {
        label: '数据七',
        data: [50, 30, 40, 21, 46, 15, 28]
      }]
    }
  },
```

```
  components: {
    ddChart
  }
}
</script>
```

效果如图 11-10 所示。

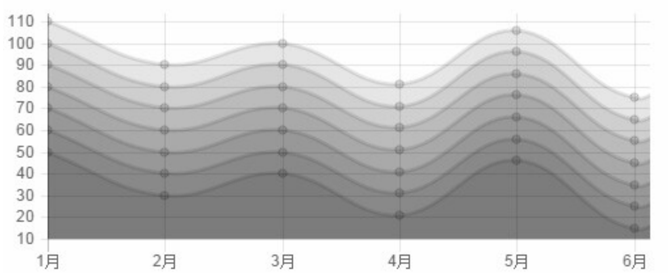


图11-10　chart

11.5　常见问题解析

1. camelCase & kebab-case

HTML 标签中的属性名不区分大小写。设置 prop 名字为 camelCase 形式的时候，需要转换为 kebab-case 形式（短横线隔开）在 HTML 中使用。代码示例如下：

```
Vue.component('child', {
  // 这里可以是 camelCase 形式
  props: ['myMessage'],
  template: '<span>{{ myMessage }}</span>'
})
<!-- 对应在 HTML 中必须是短横线分隔-->
<child my-message="hello!"></child>
```

2. 字面量语法&动态语法

初学者常犯的一个错误是使用字面量语法传递数值。代码示例如下：

```
<!-- 传递了一个字符串 "1" -->
<comp some-prop="1"></comp>
```

因为它是一个字面 prop，它的值是字符串"1"，而不是以实际的数字传下去。如果想传递一个真实的 JavaScript 类型的数字，则需要使用动态语法，从而让它的值被当作 JavaScript 表达式

计算。代码示例如下：

```
<!-- 传递实际的数字 -->
<comp :some-prop="1"></comp>
```

3. 组件选项问题

传入 Vue 构造器的多数选项也可以用在 Vue.extend()中，不过有两个特例：data 和 el。试想，如果简单地把一个对象作为 data 选项传给 Vue.extend()，代码示例如下：

```
var data = { a: 1 }
var MyComponent = Vue.extend({
  data: data
})
```

这样做的问题是 MyComponent 所有的实例将共享同一个 data 对象！这基本不是我们想要的，因此应当使用一个函数作为 data 选项，让这个函数返回一个新对象。代码示例如下：

```
var MyComponent = Vue.extend({
  data: function () {
    return { a: 1 }
  }
})
```

4. 模板解析

Vue 的模板是 DOM 模板，使用浏览器原生的解析器而不是自己实现一个。相比字符串模板，DOM 模板有一些好处，但是也有问题，它必须是有效的 HTML 片段。一些 HTML 元素对什么元素可以放在它里面有限制。常见的限制有：

- a 不能包含其他的交互元素（如按钮、链接）。
- ul 和 ol 只能直接包含 li。
- select 只能包含 option 和 optgroup。
- table 只能直接包含 thead、tbody、tfoot、tr、caption、col、colgroup。
- tr 只能直接包含 th 和 td。

在实际应用中，这些限制会导致意外的结果。尽管在简单的情况下它可能可以工作，但是我们不能依赖自定义组件在浏览器验证之前的展开结果。例如<my-select><option>...</option></my-select>不是有效的模板，即使 my-select 组件最终展开为<select>...</select>。

另一个结果是，自定义标签（包括自定义元素和特殊标签，如<component>、<template>、<partial>）不能用在 ul、select、table 等对内部元素有限制的标签内。放在这些元素内部的自定义标签将被提到元素的外面，因而渲染不正确。

自定义元素应当使用 is 特性，代码示例如下：

```
<table>
  <tr is="my-component"></tr>
</table>
```

<template>不能用在<table>内，这时应使用<tbody>，<table>可以有多个<tbody>。代码示例如下：

```
<table>
  <tbody v-for="item in items">
    <tr>Even row</tr>
    <tr>Odd row</tr>
  </tbody>
</table>
```

5. 动态组件与异步组件结合

上面的章节中分别讲述了动态组件和异步组件，这里简单展示一下它们如何结合且可以像正常组件一样，通过 props 传递数据（其中 template 用了两种形式展现，只是为了表达它有多种展现方式，如何使用需要针对业务场景来区分，此处只是为了演示）。代码示例如下：

```
<body id="example">
  <input type="radio" id="one" value="fast" v-model="currentView">
  <label for="one">fast</label>
  <br>
  <input type="radio" id="two" value="bus" v-model="currentView">
  <label for="two">bus</label>
  <br>
  <input type="radio" id="two" value="business" v-model="currentView">
  <label for="two">business</label>
  <template id="business">
    <div>{{msg}}滴滴专车</div>
  </template>
  <template id="fast">
    <div>{{msg}}滴滴快车</div>
  </template>
  <component :is="currentView" :msg="hello"></component>
</body>
```

```
<script>
  var bus = Vue.component('bus', function (resolve, reject) {
    setTimeout(function () {
      resolve({
        props: ['msg'],
        template: '<div @click="show">{{msg}} 滴滴巴士</div>',
        data: function(){ return {} },
        methods: {
          show: function(){
            alert("haha~")
          }
        }
      })
    }, 100);
  })

  var business = Vue.extend({
    // 声明 props
    props: ['msg'],
    template: '#business',
    replace: true
  })
  var fast = Vue.extend({
    // 声明 props
    props: ['msg'],
    template: '#fast',
    replace: true
  })
  // 创建根实例
  new Vue({
    el: '#example',
    data: {
      currentView: 'fast',
      hello:'hi'
    },
    components: {
      fast: fast,
      bus: bus,
      business: business
    }
  })
</script>
```

6. 如何解决数据层级结构太深的问题

在开发业务时，经常会出现异步获取数据的情况，有时候数据层次比较深。代码示例如下：

```
<span class="airport"
  v-text="ticketInfo.flight.fromSegments[ticketInfo.flight.fromSegments.length -
1].depAirportZh">
</span>
```

我们可以使用 vm.$set 手动定义一层数据，代码示例如下：

```
vm.$set("depAirportZh"  ,ticketInfo.flight.fromSegments[ticketInfo.flight.fromSegment
s.length - 1].depAirportZh)
```

现在我们来看一下$set 方法。

参数：

```
{String} keypath
{*} value
```

用法：

设置 Vue 实例的属性值。在多数情况下应当使用普通对象语法，如 vm.a.b = 123。这个方法只用于下面情况：

- 使用 keypath 动态地设置属性。
- 设置不存在的属性。

如果 keypath 不存在，将递归地创建并建立追踪。如果用它创建一个顶级属性，实例将被强制进入“digest 循环”，在此过程中重新计算所有的 Watcher。代码示例如下：

```
var vm = new Vue({
  data: {
    a: {
      b: 1
    }
  }
})

// keypath 存在
vm.$set('a.b', 2)
vm.a.b // -> 2

// keypath 不存在
```

```
vm.$set('c', 3)
vm.c // -> 3
```

7. 后端数据交互

如果配合 vue-resource，代码示例如下：

```
new Vue({
  el: '#app',
  data: {
    todos: [ ]
  },
  created: function() {
    this.$http
      .get('我们的代码体')
      .then((data) => {
         this.todos = data;
      })
  }
});
```

如果配合 jQuery 的 AJAX，代码示例如下：

```
new Vue({
  el: '#app',
  data: {
    todos: [ ]
  },
  created: function() {
    $.get('我们的 API')
      .done((data) => {
        this.todos = data;
      });
  }
});
```

8. data 中没有定义计算属性，它是如何被使用的

代码示例如下：

```
<div id="example">
  a={{ a }}, b={{ b }}
</div>

var vm = new Vue({
  el: '#example',
```

```
  data: {
    a: 1
  },
  computed: {
    b: function () {
      return this.a + 1
    }
  }
})
```

对于上面计算属性 b 是怎么被使用的，源码定义如下：

```
<!--源码目录：src\instance\internal\state.js 207 行-->
Vue.prototype._initComputed = function () {
  var computed = this.$options.computed
  if (computed) {
    for (var key in computed) {
      var userDef = computed[key]
      var def = {
        enumerable: true,
        configurable: true
      }
      if (typeof userDef === 'function') {
        def.get = makeComputedGetter(userDef, this)
        def.set = noop
      } else {
        def.get = userDef.get
          ? userDef.cache !== false
            ? makeComputedGetter(userDef.get, this)
            : bind(userDef.get, this)
          : noop
        def.set = userDef.set
          ? bind(userDef.set, this)
          : noop
      }
      Object.defineProperty(this, key, def)
    }
  }
}
```

这里并没有把计算数据放到$data 里面去，而是通过 Object.defineProperty(this, key, def)直接定义到了实例上。

第 12 章

表单校验

表单校验在 Web 应用中是很重要的一环，在 Vue.js 应用中，我们可以使用 vue-validator 2.1.3 插件对表单进行校验，以下讲解均基于 vue-validator 2.1.3 版本，后面介绍将不加版本号了。

12.1 安装

vue-validator 提供了 npm、bower、手动编译等安装方式，可以根据业务需要选择其中一种方式进行安装。下面介绍 npm 和手动编译两种安装方式的安装方法及适合场景。

1. npm

当业务代码使用 Webpack 等支持 CommonJS 规范的模块化打包器来构建时，可以使用 npm 包的方式来安装：

```
$ npm install vue-validator
```

因为 vue-validator 是 Vue.js 的一个插件，所以 vue-validator 需要使用 Vue.use(PluginContructor)（Vue.js 用此方法来注册插件）注册到 Vue 对象上，在 vue-validator 内部会检测 window.Vue 对象是否存在，如果存在则会自动调用 Vue.use() 方法；否则需要使用者手动调用 Vue.use(VueValidator)来确保校验插件注册到 Vue 中。在 Webpack 等支持 CommonJS 规范的环境中，Vue 对象并不会暴露到全局 window 对象中，而是会通过 module.exports 形式输出，因此需要使用者手动注册。代码示例如下：

```
var Vue = require('vue');
var VueValidator = require('vue-validator');
// 安装 vue-validator
Vue.use(VueValidator);
```

2. 手动编译

当想尝试一些 Vue 中并未发布的新特性时，可以直接 clone 源码，手动构建来实现。由于在未正式发布之前，有些特性可能会被移除，所以不建议在生产环境中使用手动编译方式安装。执行以下命令：

```
$ git clone https://github.com/vuejs/vue-validator.git node_modules/vue-validator
$ cd node_modules/vue-validator
$ npm install
$ npm run build
```

12.2　基本使用

为了快速入门，我们来看一个完整的例子。代码示例如下：

```
new Vue({
  el: '#app'
})

<div id="app">
  <validator name="validation">
    <form novalidate>
      <div class="username-field">
        <label for="username">username:</label>
        <input id="username" type="text" v-validate:username="['required']">
      </div>
      <div class="comment-field">
        <label for="comment">comment:</label>
        <input id="comment" type="text" v-validate:comment="{ maxlength: 256 }">
      </div>
      <div class="errors">
        <p v-if="$validation.username.required">Required your name.</p>
        <p v-if="$validation.comment.maxlength">Your comment is too long.</p>
      </div>
      <input type="submit" value="send" v-if="$validation.valid">
    </form>
  </validator>
</div>
```

我们可以看到要校验的表单元素包裹在 validator 自定义元素指令中，而在要校验的表单控

件元素的 v-validate 属性上绑定相应的校验规则。验证结果会保存在组件实例的$validation 属性下，$validation 是由 validator 元素的 name 属性和$前缀组成的。

注：validator 元素的 name 属性请不要使用 Vue.js 中内置的属性名称，如$event。

12.3 验证结果结构

在 12.2 节中，我们了解到 vue-validator 将验证结果存储在组件实例上以 validator 元素 name 属性值及$前缀为键名的属性下。下面我们来看一下验证结果的结构。代码示例如下：

```
{
  // 表单整体验证结果
  valid: true,
  invalid: false,
  touched: false,
  undefined: true,
  dirty: false,
  pristine: true,
  modified: false,
  errors: [{
    field: 'field1', validator: 'required', message: 'required field1'
  }, ... {
    field: 'fieldX', validator: 'customValidator', message: 'invalid fieldX'
  }],
  // 字段一验证结果
  field1: {
    required: false,
    // vue-validator 内置校验器，表示是否已填写，true 表示校验通过，false 表示校验失败
    email: true, // 自定义校验器
    url: 'invalid url format', // 自定义校验器
    ...
    customValidator1: false, // 自定义校验器
    // 校验状态属性
    valid: false,
    invalid: true,
    touched: false,
    undefined: true,
    dirty: false,
```

```
    pristine: true,
    modified: false,
    errors: [{
      validator: 'required', message: 'required field1'
    }]
  },
  ...
  // 字段 x 验证结果
  fieldX: {
    min: false, // 内置校验器
    ...
    customValidator: true,
    // 校验状态属性
    valid: false,
    invalid: true,
    touched: true,
    undefined: false,
    dirty: true,
    pristine: false,
    modified: true,
    errors: [{
      validator: 'customValidator', message: 'invalid fieldX'
    }]
  }
}
```

从以上校验结果的结构中我们可以了解到，校验结果由两部分组成，即表单整体校验结果和单个字段校验结果。

单个字段校验结果包括以下校验属性：

- Valid —— 字段校验是否通过，通过返回 true，失败返回 false。
- invalid —— valid 取反。
- touched —— 校验字段所在元素获得过焦点时返回 true，否则返回 false。
- untouched —— touched 取反。
- modified —— 当元素值与初始值不同时返回 true，否则返回 false。

- dirty —— 字段值改变过至少一次返回 true，否则返回 false。
- pristine —— dirty 取反。
- errors —— 如果校验没通过，则返回错误字段信息数组，否则返回 undefined。

表单整体校验结果包括以下校验属性：

- valid —— 所有字段校验是否通过，通过返回 true，失败返回 false。
- invalid —— valid 取反。
- touched —— 只要有一个校验字段所在元素获得过焦点就返回 true，否则返回 false。
- untouched —— touched 取反。
- modified —— 只要有一个字段对应元素值与初始值不同就返回 true，否则返回 false。
- dirty —— 只要有一个校验字段对应元素值改变过至少一次就返回 true，否则返回 false。
- pristine —— dirty 取反。
- errors —— 如果整体校验没通过，则返回错误字段信息数组，否则返回 undefined。

12.4 验证器语法

v-validate 指令语法如下：

```
v-validate[:field]="array literal | object literal | binding"
```

下面我们来了解一下 field 的含义，以及 v-validate 校验规则的格式。

12.4.1 校验字段名 field

field 用来标识校验字段，之后可以用该字段来引用校验结果。

在 vue-validator 版本小于等于 2.0-alpha 时，校验时依赖 v-model 指令。代码示例如下：

```
<!-- ~1.4.4 版本校验语法-->
<form novalidate>
  <input type="text" v-model="comment" v-validate="minLength: 16, maxLength: 128">
  <div>
```

```
    <span v-show="validation.comment.minLength">Your comment is too short.</span>
    <span v-show="validation.comment.maxLength">Your comment is too long.</span>
  </div>
  <input type="submit" value="send" v-if="valid">
</form>
```

在 2.0-alpha 及以后版本中，使用 v-validate 指令进行校验。代码示例如下：

```
<!-- 2.0-alpha 及以后版本校验语法 -->
<validator name="validation">
  <form novalidate>
    <input type="text" v-validate:comment="{ minlength: 16, maxlength: 128 }">
    <div>
      <span v-show="$validation.comment.minlength">Your comment is too short.</span>
      <span v-show="$validation.comment.maxlength">Your comment is too long.</span>
    </div>
    <input type="submit" value="send" v-if="valid">
  </form>
</validator>
```

和指令命名规则一样，在 HTML 中我们可以用连字符（-）来命名 field，如以下例子中的 user-name，然后通过驼峰式命名法来引用它。代码示例如下：

```
<validator name="validation">
  <form novalidate>
    <input type="text" v-validate:user-name="{ minlength: 16 }">
    <div>
      <span v-if="$validation.userName.minlength">Your user name is too short.</span>
    </div>
  </form>
</validator>
```

当需要动态绑定校验字段的名称时，我们可以在要校验的元素上使用 field 属性。代码示例如下：

```
<div id="app">
  <validator name="validation">
    <form novalidate>
      <p class="validate-field" v-for="field in fields">
      <label :for="field.id">{{field.label}}</label>
     <input type="text" :id="field.id" :placeholder="field.placeholder" field="{{field.
      name}}" v-validate="field.validate">
      </p>
```

```
    <pre>{{ $validation | json }}</pre>
  </form>
 </validator>
</div>
new Vue({
  el: '#app',
  data: {
    fields: [{
      id: 'username',
      label: 'username',
      name: 'username',
      placeholder: 'input your username',
      validate: { required: true, maxlength: 16 }
    }, {
      id: 'message',
      label: 'message',
      name: 'message',
      placeholder: 'input your message',
      validate: { required: true, minlength: 8 }
    }]
  }
})
```

12.4.2 校验规则定义

v-validate 指令用来定义校验规则，其值可以是数组字面量、对象字面量、组件实例数据属性名。

1. 数组字面量

当校验器不需要额外参数时，我们可以使用数组字面量的形式。如 required 校验器，只要出现就代表该校验器所在元素是必填项。代码示例如下：

```
<validator name="validation">
  <form novalidate>
    Zip: <input type="text" v-validate:zip="['required']"><br />
    <div>
      <span v-if="$validation.zip.required">Zip code is required.</span>
    </div>
  </form>
</validator>
```

2. 对象字面量

对象字面量语法适合需要额外参数的校验器。如限制输入长度的校验器 minlength，需要说明最小长度限制是多少。代码示例如下：

```
<validator name="validation">
  <form novalidate>
    ID: <input type="text" v-validate:id="{ required: true, minlength: 3, maxlength: 16 }"><br />
    <div>
      <span v-if="$validation.id.required">ID is required</span>
      <span v-if="$validation.id.minlength">Your ID is too short.</span>
      <span v-if="$validation.id.maxlength">Your ID is too long.</span>
    </div>
  </form>
</validator>
```

我们还可以使用对象字面量语法通过 rule 字段来自定义验证规则。代码示例如下：

```
<validator name="validation">
  <form novalidate>
    ID: <input type="text" v-validate:id="{ minlength: { rule: 3 }, maxlength: { rule: 16 } }"><br />
    <div>
      <span v-if="$validation.id.minlength">Your ID is too short.</span>
      <span v-if="$validation.id.maxlength">Your ID is too long.</span>
    </div>
  </form>
</validator>
```

3. 实例数据属性

v-validate 的值可以是组件实例的数据属性，这样可以用来动态绑定校验规则。代码示例如下：

```
new Vue({
  el: '#app',
  data: {
    rules: {
      minlength: 3,
      maxlength: 16
```

```
    }
  }
})
<div id="app">
  <validator name="validation">
    <form novalidate>
      ID: <input type="text" v-validate:id="rules"><br />
      <div>
        <span v-if="$validation.id.minlength">Your ID is too short.</span>
        <span v-if="$validation.id.maxlength">Your ID is too long.</span>
      </div>
      </form>
  </validator>
</div>
```

4. 与 terminal 指令同时使用

当与 v-if、v-for 等 terminal 指令同时使用时，需要把可验证的目标元素包裹在<template>之类的不可见标签内。因为 v-validate 指令不能与 terminal 指令同时使用，所以包裹在了 div 元素中。代码示例如下：

```
new Vue({
  el: '#app',
  data: {
    enable: true
  }
})
<div id="app">
  <validator name="validation">
    <form novalidate>
      <div class="username">
        <label for="username">username:</label>
        <input id="username" type="text"
              @valid="this.enable = true"
              @invalid="this.enable = false"
              v-validate:username="['required']">
      </div>
      <div v-if="enable" class="password">
        <label for="password">password:</label>
        <input id="password" type="password" v-validate:password="{
```

```
        required: { rule: true }, minlength: { rule: 8 }
      }"/>
    <div>
  </form>
 </validator>
</div>
```

12.5　内置验证规则

为了提高开发效率，vue-validator 为我们提供了一些常用的内置的验证规则，如图 12-1 所示。

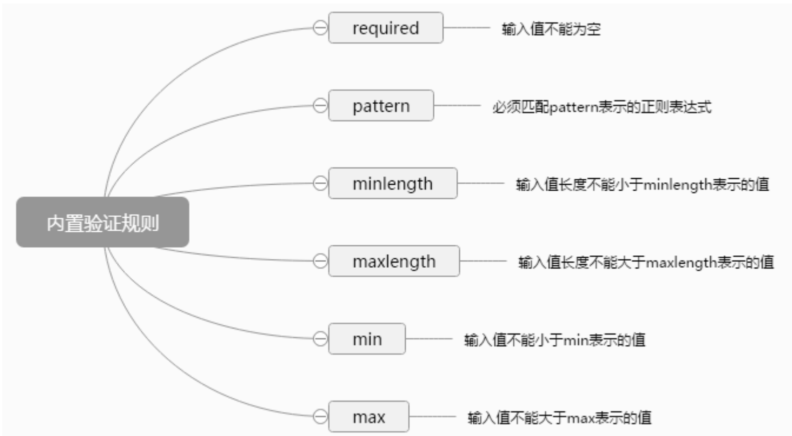


图12-1　内置验证规则

12.5.1　required

必填校验器，该校验器用来校验字段值是否不为空。

可以使用该校验器的元素有：

- input[type="text"]
- input[type="radio"]
- input[type="checkbox"]
- input[type="number"]

- input[type="password"]
- input[type="email"]
- input[type="tel"]
- input[type="url"]
- select
- textarea

源码定义如下：

```
<!--源码目录：src/validators.js -->
export function required (val) {
  if (Array.isArray(val)) {
    if (val.length !== 0) {
      let valid = true
      for (let i = 0, l = val.length; i < l; i++) {
        valid = required(val[i])
        if (!valid) {
          break
        }
      }
      return valid
    } else {
      return false
    }
  } else if (typeof val === 'number' || typeof val === 'function') {
    return true
  } else if (typeof val === 'boolean') {
    return val
  } else if (typeof val === 'string') {
    return val.length > 0
  } else if (val !== null && typeof val === 'object') {
    return Object.keys(val).length > 0
  } else if (val === null || val === undefined) {
    return false
  }
}
```

12.5.2　pattern

正则匹配校验器，校验元素值是否匹配 pattern 所表示的正则表达式。

可以使用该校验器的元素有：

- input[type="text"]
- input[type="number"]
- input[type="password"]
- input[type="email"]
- input[type="tel"]
- input[type="url"]
- textarea

源码定义如下：

```
<!--源码目录：src/validators.js -->
export function pattern (val, pat) {
  if (typeof pat !== 'string') { return false }
  let match = pat.match(new RegExp('^/(.*?)/([gimy]*)$'))
  if (!match) { return false }
  return new RegExp(match[1], match[2]).test(val)
}
```

12.5.3　minlength

最小长度校验器，校验元素的长度是否大于 minlength。

可以使用该校验器的元素有：

- input[type="text"]
- input[type="checkbox"]
- input[type="number"]

- input[type="password"]
- input[type="email"]
- input[type="tel"]
- input[type="url"]
- select
- textarea

源码定义如下：

```
<!--源码目录：src/validators.js -->
export function minlength (val, min) {
  if (typeof val === 'string') {
    return isInteger(min, 10) && val.length >= parseInt(min, 10)
  } else if (Array.isArray(val)) {
    return val.length >= parseInt(min, 10)
  } else {
    return false
  }
}
```

12.5.4 maxlength

最大长度校验器，校验元素的长度是否小于 maxlength。

可以使用该校验器的元素有：

- input[type="text"]
- input[type="checkbox"]
- input[type="number"]
- input[type="password"]
- input[type="email"]
- input[type="tel"]

- input[type="url"]
- select
- textarea

源码定义如下：

```
<!--源码目录：src/validators.js -->
export function maxlength (val, max) {
  if (typeof val === 'string') {
    return isInteger(max, 10) && val.length <= parseInt(max, 10)
  } else if (Array.isArray(val)) {
    return val.length <= parseInt(max, 10)
  } else {
    return false
  }
}
```

12.5.5　min

最小长度校验器，校验元素的值是否大于 min 的值。

可以使用该校验器的元素有：

- input[type="text"]
- input[type="number"]
- textarea

源码定义如下：

```
<!--源码目录：src/validators.js -->
export function min (val, arg) {
  return !isNaN(+(val)) && !isNaN(+(arg)) && (+(val) >= +(arg))
}
```

12.5.6　max

最大长度校验器，校验元素的值是否小于 max 的值。

可以使用该校验器的元素有：

- input[type="text"]
- input[type="number"]
- textarea

源码定义如下：

```
<!--源码目录：src/validators.js -->
export function max (val, arg) {
  return !isNaN(+(val)) && !isNaN(+(arg)) && (+(val) <= +(arg))
}
```

12.6 与 v-model 同时使用

vue-validator 会自动校验通过 v-model 动态设置的值。代码示例如下：

```
<div id="app">
  <validator name="validation1">
    <form novalidate>
      message: <input type="text" v-model="msg" v-validate:message="{ required: true,
minlength: 8 }"><br />
      <div>
        <p v-if="$validation1.message.required">Required your message.</p>
        <p v-if="$validation1.message.minlength">Too short message.</p>
      </div>
    </form>
  </validator>
</div>
var vm = new Vue({
  el: '#app',
  data: {
    msg: ''
  }
})
setTimeout(function () {
  vm.msg = 'hello world!!'
}, 2000)
```

12.7　重置校验结果

我们可以通过在 Vue 组件实例上调用$resetValidation()方法来动态重置校验结果。代码示例如下：

```
<div id="app">
  <validator name="validation1">
    <form novalidate>
      <div class="username-field">
        <label for="username">username:</label>
        <input id="username" type="text" v-validate:username="['required']">
      </div>
      <div class="comment-field">
        <label for="comment">comment:</label>
        <input id="comment" type="text" v-validate:comment="{ maxlength: 256 }">
      </div>
      <div class="errors">
        <p v-if="$validation1.username.required">Required your name.</p>
        <p v-if="$validation1.comment.maxlength">Your comment is too long.</p>
      </div>
      <input type="submit" value="send" v-if="$validation1.valid">
      <button type="button" @click="onReset">Reset Validation</button>
    </form>
    <pre>{{ $validation1 | json }}</pre>
  </validator>
</div>
new Vue({
  el: '#app',
  methods: {
    onReset: function () {
      this.$resetValidation()
    }
  }
})
```

12.8　表单元素

由于表单元素 checkbox、radio、select 等的特殊性，我们有必要单独介绍一下 vue-validator

在这些表单元素上的使用。

复选框 checkbox：

```
<div id="app">
  <validator name="validation1">
    <form novalidate>
      <h1>Survey</h1>
      <fieldset>
        <legend>Which do you like fruit ?</legend>
        <input id="apple" type="checkbox" value="apple" v-validate:fruits="{
          required: { rule: true, message: requiredErrorMsg },
          minlength: { rule: 1, message: minlengthErrorMsg },
          maxlength: { rule: 2, message: maxlengthErrorMsg }
        }">
        <label for="apple">Apple</label>
        <input id="orange" type="checkbox" value="orange" v-validate:fruits>
        <label for="orange">Orage</label>
        <input id="grape" type="checkbox" value="grape" v-validate:fruits>
        <label for="grape">Grape</label>
        <input id="banana" type="checkbox" value="banana" v-validate:fruits>
        <label for="banana">Banana</label>
        <ul class="errors">
          <li v-for="msg in $validation1.fruits.errors">
            <p>{{msg}}</p>
          </li>
        </ul>
      </fieldset>
    </form>
  </validator>
</div>
new Vue({
  el: '#app',
  computed: {
    requiredErrorMsg: function () {
      return 'Required fruit !!'
    },
    minlengthErrorMsg: function () {
      return 'Please chose at least 1 fruit !!'
    },
```

```
    maxlengthErrorMsg: function () {
      return 'Please chose at most 2 fruits !!'
    }
  }
})
```

单选钮 radio

```
<div id="app">
  <validator name="validation1">
    <form novalidate>
      <h1>Survey</h1>
      <fieldset>
        <legend>Which do you like fruit ?</legend>
        <input id="apple" type="radio" name="fruit" value="apple" v-validate:fruits="{
          required: { rule: true, message: requiredErrorMsg }
        }">
        <label for="apple">Apple</label>
        <input id="orange" type="radio" name="fruit" value="orange" v-validate:fruits>
        <label for="orange">Orage</label>
        <input id="grape" type="radio" name="fruit" value="grape" v-validate:fruits>
        <label for="grape">Grape</label>
        <input id="banana" type="radio" name="fruit" value="banana" v-validate:fruits>
        <label for="banana">Banana</label>
        <ul class="errors">
          <li v-for="msg in $validation1.fruits.errors">
            <p>{{msg}}</p>
          </li>
        </ul>
      </fieldset>
    </form>
  </validator>
</div>
new Vue({
  el: '#app',
  computed: {
    requiredErrorMsg: function () {
      return 'Required fruit !!'
    }
  }
})
```

下拉列表 select：

```
<div id="app">
  <validator name="validation1">
    <form novalidate>
      <select v-validate:lang="{ required: true }">
        <option value="">----- select your favorite programming language -----</option>
        <option value="javascript">JavaScript</option>
        <option value="ruby">Ruby</option>
        <option value="python">Python</option>
        <option value="perl">Perl</option>
        <option value="lua">Lua</option>
        <option value="go">Go</option>
        <option value="rust">Rust</option>
        <option value="elixir">Elixir</option>
        <option value="c">C</option>
        <option value="none">Not listed here</option>
      </select>
      <div class="errors">
        <p v-if="$validation1.lang.required">Required !!</p>
      </div>
    </form>
  </validator>
</div>
new Vue({ el: '#app' })
```

12.9 各校验状态对应的 class

为了提升用户体验，有时候我们需要基于校验结果来对表单元素应用不同的样式。vue-validator 会基于校验结果自动在元素上添加一些校验状态的 class。代码示例如下：

```
<input id="username" type="text" v-validate:username="{
  required: { rule: true, message: 'required you name !!' }
}">
```

在页面初始化后，以上代码的输出结果类似于：

```
<input id="username" type="text" class="invalid untouched pristine">
```

各校验状态对应的 class 如表 12-1 所示。

表 12–1　各校验状态对应的 class

校验状态	对应 class（默认）
valid	valid
invalid	invalid
touched	touched
untouched	untouched
pristine	pristine
dirty	dirty
modified	modified

12.9.1　自定义校验状态 class

有时候我们可能需要使用自定义的状态 class，这时我们可以通过 classes 属性来自定义各状态对应的 class。代码示例如下：

```
<validator name="validation1"
         :classes="{ touched: 'touched-validator', dirty: 'dirty-validator' }">
  <label for="username">username:</label>
  <input id="username"
        type="text"
        :classes="{ valid: 'valid-username', invalid: 'invalid-username' }"
        v-validate:username="{ required: { rule: true, message: 'required you name !!' } }">
</validator>
```

注：classes 属性只能在 validator 元素或应用了 v-validate 的元素上使用有效。

12.9.2　在其他元素上使用校验状态 class

一般校验状态 class 会被添加到 v-validate 指令的元素上，有时候我们需要将 class 添加到 v-validate 指令所在元素的包裹元素上，这时我们可以在相应的包裹元素上使用 v-validate-class 指令。代码示例如下：

```
<validator name="validation1"
         :classes="{ touched: 'touched-validator', dirty: 'dirty-validator' }">
  <div v-validate-class class="username">
    <label for="username">username:</label>
```

```
    <input id="username"
           type="text"
           :classes="{ valid: 'valid-username', invalid: 'invalid-username' }"
           v-validate:username="{ required: { rule: true, message: 'required you name !!' }
    }">
  </div>
</validator>
```

在页面初始化后，以上代码的输出结果为：

```
<div class="username invalid-username untouched pristine">
  <label for="username">username:</label>
  <input id="username" type="text">
</div>
```

12.10 分组校验

vue-validator 支持分组校验。如输入密码时，我们可以将第一次输入密码与再次输入密码确认放入同一个校验组内，只有两次校验都通过了，该组校验才算通过。代码示例如下：

```
<validator name="validation1" :groups="['passwordGroup']">
  username: <input type="text" v-validate:username="['required']"><br>
  password: <input type="password" group="passwordGroup" v-validate:password="{ minlength: 8,
required: true }"/><br>
  confirm password: <input type="password" group="passwordGroup" v-validate:password-
confirm="{ minlength: 8, required: true }"/><br>
  <span v-if="$validation1.username.invalid">Invalid username!</span><br>
  <span v-if="$validation1.passwordGroup.invalid">Invalid password input!</span>
</validator>
```

以上代码只有当 password 规则和 password-confirm 规则都校验通过后，passwordGroup.valid 才会返回 true。

12.11 错误信息

在校验失败时，我们需要获得相应的错误提示信息，错误信息可以直接存储在校验规则中。代码示例如下：

```
<validator name="validation1">
  <div class="username">
```

```
    <label for="username">username:</label>
    <input id="username" type="text" v-validate:username="{
      required: { rule: true, message: 'required you name !!' }
    }">
    <span v-if="$validation1.username.required">
      {{ $validation1.username.required }}
    </span>
  </div>
  <div class="password">
    <label for="password">password:</label>
    <input id="password" type="password" v-validate:password="{
      required: { rule: true, message: 'required you password !!' },
      minlength: { rule: 8, message: 'your password short too !!' }
    }"/>
    <span v-if="$validation1.password.required">
      {{ $validation1.password.required }}
    </span>
    <span v-if="$validation1.password.minlength">
      {{ $validation1.password.minlength }}
    </span>
  </div>
</validator>
```

1．输出所有错误信息

如果需要输出当前所有校验失败的错误信息，则可以通过 v-for 指令来循环 errors 数组。代码示例如下：

```
<validator name="validation1">
  <div class="username">
    <label for="username">username:</label>
    <input id="username" type="text" v-validate:username="{
      required: { rule: true, message: 'required you name !!' }
    }">
  </div>
  <div class="password">
      <label for="password">password:</label>
      <input id="password" type="password" v-validate:password="{
      required: { rule: true, message: 'required you password !!' },
      minlength: { rule: 8, message: 'your password short too !!' }
    }"/>
```

```
    </div>
    <div class="errors">
    <ul>
      <li v-for="error in $validation1.errors">
        <p>{{error.field}}: {{error.message}}</p>
      </li>
    </ul>
  </div>
</validator>
```

2. 输出单个校验错误信息

我们可以输出单个校验规则或者单个分组的错误信息。代码示例如下：

```
<div id="app">
  <validator :groups="['profile', 'password']" name="validation1">
    <div class="username">
      <label for="username">username:</label>
      <input id="username" type="text" group="profile" v-validate:username="{
        required: { rule: true, message: 'required you name !!' }
      }">
    </div>
    <div class="url">
      <label for="url">url:</label>
      <input id="url" type="text" group="profile" v-validate:url="{
        required: { rule: true, message: 'required you name !!' },
        url: { rule: true, message: 'invalid url format' }
      }">
    </div>
    <div class="old">
      <label for="old">old password:</label>
      <input id="old" type="password" group="password" v-validate:old="{
        required: { rule: true, message: 'required you old password !!' },
        minlength: { rule: 8, message: 'your old password short too !!' }
      }"/>
    </div>
    <div class="new">
      <label for="new">new password:</label>
      <input id="new" type="password" group="password" v-validate:new="{
        required: { rule: true, message: 'required you new password !!' },
        minlength: { rule: 8, message: 'your new password short too !!' }
```

```
      }"/>
    </div>
    <div class="confirm">
      <label for="confirm">confirm password:</label>
      <input id="confirm" type="password" group="password" v-validate:confirm="{
        required: { rule: true, message: 'required you confirm password !!' },
        minlength: { rule: 8, message: 'your confirm password short too !!' }
      }"/>
    </div>
    <div class="errors">
      <validator-errors group="profile" :validation="$validation1">
      </validator-errors>
    </div>
  </validator>
</div>
Vue.validator('url', function (val) {
  return /^(http\:\/\/|https\:\/\/)(.{4,})$/.test(val)
})
new Vue({ el: '#app' })
```

12.11.1　错误信息输出组件

在上节示例中，我们使用 v-for 指令来遍历输出错误信息。在 vue-validator 中，可以使用 validator-errors 组件来更便捷地输出错误信息。代码示例如下：

```
  <validator name="validation1">
    <div class="username">
    <label for="username">username:</label>
<input id="username" type="text" v-validate:username="{
      required: { rule: true, message: 'required you name !!' }
    }">
    </div>
    <div class="password">
      <label for="password">password:</label>
      <input id="password" type="password" v-validate:password="{
      required: { rule: true, message: 'required you password !!' },
      minlength: { rule: 8, message: 'your password short too !!' }
    }"/>
  </div>
```

```
  <div class="errors">
    <validator-errors :validation="$validation1"></validator-errors>
  </div>
</validator>
```

在页面初始化后，以上代码的输出结果类似于：

```
<div class="username">
  <label for="username">username:</label>
  <input id="username" type="text">
</div>
<div class="password">
  <label for="password">password:</label>
  <input id="password" type="password">
</div>
<div class="errors">
  <div>
    <p>password: your password short too !!</p>
  </div>
  <div>
    <p>password: required you password !!</p>
  </div>
  <div>
    <p>username: required you name !!</p>
  </div>
</div>
```

如果不喜欢默认的错误信息模板，我们还可以自定义错误信息模板，validator-errors 提供了 Component 模板、Partial 模板两种定义方式。

1. Component 模板

```
<div id="app">
  <validator name="validation1">
    <div class="username">
      <label for="username">username:</label>
      <input id="username" type="text" v-validate:username="{
        required: { rule: true, message: 'required you name !!' }
      }">
    </div>
    <div class="password">
```

```
      <label for="password">password:</label>
      <input id="password" type="password" v-validate:password="{
        required: { rule: true, message: 'required you password !!' },
        minlength: { rule: 8, message: 'your password short too !!' }
      }"/>
    </div>
    <div class="errors">
      <validator-errors :component="'custom-error'" :validation="$validation1">
      </validator-errors>
    </div>
  </validator>
</div>
Vue.component('custom-error', {
  props: ['field', 'validator', 'message'],
  template: '<p class="error-{{field}}-{{validator}}">{{message}}</p>'
})
new Vue({ el: '#app' })
```

2. Partial 模板

```
<div id="app">
  <validator name="validation1">
    <div class="username">
      <label for="username">username:</label>
      <input id="username" type="text" v-validate:username="{
        required: { rule: true, message: 'required you name !!' }
      }">
    </div>
    <div class="password">
      <label for="password">password:</label>
      <input id="password" type="password" v-validate:password="{
        required: { rule: true, message: 'required you password !!' },
        minlength: { rule: 8, message: 'your password short too !!' }
      }"/>
    </div>
    <div class="errors">
      <validator-errors partial="myErrorTemplate" :validation="$validation1">
      </validator-errors>
    </div>
  </validator>
</div>
```

```
Vue.partial('myErrorTemplate', '<p>{{field}}: {{validator}}: {{message}}</p>')
new Vue({ el: '#app' })
```

12.11.2　动态设置错误信息

在服务端验证失败时，我们可以用组件实例方法$setValidationErrors 来动态设置错误信息。代码示例如下：

```
<div id="app">
  <validator name="validation">
    <div class="username">
      <label for="username">username:</label>
      <input id="username" type="text" v-model="username" v-validate:username="{
        required: { rule: true, message: 'required you name !!' }
      }">
    </div>
    <div class="old">
      <label for="old">old password:</label>
      <input id="old" type="password" v-model="passowrd.old" v-validate:old="{
        required: { rule: true, message: 'required you old password !!' }
      }"/>
    </div>
    <div class="new">
      <label for="new">new password:</label>
      <input id="new" type="password" v-model="password.new" v-validate:new="{
        required: { rule: true, message: 'required you new password !!' },
        minlength: { rule: 8, message: 'your new password short too !!' }
      }"/>
    </div>
    <div class="confirm">
      <label for="confirm">confirm password:</label>
      <input id="confirm" type="password" v-validate:confirm="{
        required: { rule: true, message: 'required you confirm password !!' },
        confirm: { rule: passowd.new, message: 'your confirm password incorrect !!' }
      }"/>
    </div>
    <div class="errors">
      <validator-errors :validation="$validation"></validator-errors>
    </div>
```

```
    <button type="button" v-if="$validation.valid" @click.prevent="onSubmit">update</button>
  </validator>
</div>
new Vue({
  el: '#app',
  data: {
    id: 1,
    username: '',
    password: {
      old: '',
      new: ''
    }
  },
  validators: {
    confirm: function (val, target) {
      return val === target
    }
  },
  methods: {
    onSubmit: function () {
      var self = this
      var resource = this.$resource('/user/:id')
      resource.save({ id: this.id }, {
        username: this.username,
        password: this.new
      }, function (data, stat, req) {
        // 成功回调
        // ...
      }).error(function (data, stat, req) {
        // handle server error
        self.$setValidationErrors([
          { field: data.field, message: data.message }
        ])
      })
    }
  }
})
```

12.12 事件

在校验状态发生变化时，vue-validator 会触发相应的事件，我们可以通过 Vue.js 中的事件注册方法来注册相关事件。vue-validator 会触发单个字段校验事件和整个表单校验事件。

12.12.1 单个字段校验事件

对于每个 v-validate 所在元素，我们可以监听以下事件：

- valid —— 当字段验证结果变为有效时触发。
- invalid —— 当字段验证结果变为无效时触发。
- touched —— 当字段失去焦点时触发。
- dirty —— 当字段值首次变化时触发。
- modified —— 当字段值与初始值不同时或变回初始值时触发。

```
<div id="app">
  <validator name="validation1">
    <div class="comment-field">
      <label for="comment">comment:</label>
      <input type="text"
            @valid="onValid"
            @invalid="onInvalid"
            @touched="onTouched"
            @dirty="onDirty"
            @modified="onModified"
            v-validate:comment="['required']"/>
    </div>
    <div>
      <p>{{occuredValid}}</p>
      <p>{{occuredInvalid}}</p>
      <p>{{occuredTouched}}</p>
      <p>{{occuredDirty}}</p>
      <p>{{occuredModified}}</p>
    </div>
  </validator>
```

```
</div>
new Vue({
  el: '#app',
  data: {
    occuredValid: '',
    occuredInvalid: '',
    occuredTouched: '',
    occuredDirty: '',
    occuredModified: ''
  },
  methods: {
    onValid: function () {
      this.occuredValid = 'occured valid event'
      this.occuredInvalid = ''
    },
    onInvalid: function () {
      this.occuredInvalid = 'occured invalid event'
      this.occuredValid = ''
    },
    onTouched: function () {
      this.occuredTouched = 'occured touched event'
    },
    onDirty: function () {
      this.occuredDirty = 'occured dirty event'
    },
    onModified: function (e) {
      this.occuredModified = 'occured modified event: ' + e.modified
    }
  }
})
```

12.12.2　整个表单校验事件

除了监听单个字段的校验事件，我们也可以在 validator 元素上监听整个表单的校验事件。在 validator 元素上我们可以监听以下事件：

- valid —— 当全局验证结果变为有效时触发。
- invalid —— 当全局验证结果变为无效时触发。

- touched —— 当任意验证字段失去焦点时触发。
- dirty —— 当任意字段首次改变时触发。
- modified —— 当任意字段首次改变时或所有字段恢复初始值时触发。

```
<div id="app">
  <validator name="validation1"
           @valid="onValid"
           @invalid="onInvalid"
           @touched="onTouched"
           @dirty="onDirty"
           @modified="onModified">
    <div class="comment-field">
      <label for="username">username:</label>
      <input type="text"
           v-validate:username="['required']"/>
    </div>
    <div class="password-field">
      <label for="password">password:</label>
      <input type="password"
           v-validate:password="{ required: true, minlength: 8 }"/>
    </div>
    <div>
      <p>{{occuredValid}}</p>
      <p>{{occuredInvalid}}</p>
      <p>{{occuredTouched}}</p>
      <p>{{occuredDirty}}</p>
      <p>{{occuredModified}}</p>
    </div>
  </validator>
</div>
new Vue({
  el: '#app',
  data: {
    occuredValid: '',
    occuredInvalid: '',
    occuredTouched: '',
    occuredDirty: '',
    occuredModified: ''
```

```
  },
  methods: {
    onValid: function () {
      this.occuredValid = 'occured valid event'
      this.occuredInvalid = ''
    },
    onInvalid: function () {
      this.occuredInvalid = 'occured invalid event'
      this.occuredValid = ''
    },
    onTouched: function () {
      this.occuredTouched = 'occured touched event'
    },
    onDirty: function () {
      this.occuredDirty = 'occured dirty event'
    },
    onModified: function (modified) {
      this.occuredModified = 'occured modified event: ' + modified
    }
  }
})
```

12.13　延迟初始化

在 validator 元素上设置 lazy 属性可以延迟校验，只有在组件实例上调用$activateValidator方法才会初始化校验。当要校验的数据需要异步加载时，该特性可以避免数据加载之前出现错误提示。

当评论数据从服务端返回时，才进行校验。代码示例如下：

```
<div>
  <h1>Preview</h1>
  <p>{{comment}}</p>
  <validator lazy name="validation1">
    <input type="text" :value="comment" v-validate:comment="{ required: true, maxlength:
256 }"/>
    <span v-if="$validation1.comment.required">Required your comment</span>
    <span v-if="$validation1.comment.maxlength">Too long comment !!</span>
    <button type="button" value="save" @click="onSave" v-if="valid">
```

```
  </validator>
</div>
Vue.component('comment', {
  props: {
    id: Number,
  },
  data: function () {
    return { comment: '' }
  },
  activate: function (done) {
    var resource = this.$resource('/comments/:id');
    resource.get({ id: this.id }, function (comment, stat, req) {
      this.commont =  comment.body
      // 激活 validator
      this.$activateValidator()
      done()
    }.bind(this)).error(function (data, stat, req) {
      // 请求失败逻辑
      done()
    })
  },
  methods: {
    onSave: function () {
      var resource = this.$resource('/comments/:id');
 resource.save({ id: this.id }, { body: this.comment }, function (data, stat, req) {
        // 请求成功逻辑
      }).error(function (data, sta, req) {
        // 请求失败逻辑
      })
    }
  }
})
```

12.14 自定义验证器

除了内置的验证器，我们也可以根据需求自定义验证器。

12.14.1　注册自定义验证器

我们可以通过全局或局部两种注册方式来注册验证器。

1. 全局注册

```
// E-mail 格式验证器
Vue.validator('email', function (val) {
  return /^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)*)|(\".+\"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/.test(val)
})

new Vue({
  el: '#app'
  data: {
    email: ''
  }
})
<div id="app">
  <validator name="validation1">
    address: <input type="text" v-validate:address="['email']"><br />
    <div>
      <p v-show="$validation1.address.email">Invalid your mail address format.</p>
    </div>
  <validator>
</div>
```

2. 局部注册

可以通过组件的 validators 选项来注册只能在组件内使用的验证器。验证器方法返回 true 表示校验通过，返回 false 表示校验失败。

下例中注册了 numeric 和 url 两个自定义验证器。

```
new Vue({
  el: '#app',
  validators: { // `numeric` and `url` custom validator is local registration
    numeric: function (val/*,rule*/) {
      return /^[-+]?[0-9]+$/.test(val)
    },
    url: function (val) {
```

```
     return /^(http\:\/\/|https\:\/\/)(.{4,})$/.test(val)
    }
  },
  data: {
    email: ''
  }
})
<div id="app">
  <validator name="validation1">
    username: <input type="text" v-validate:username="['required']"><br />
    email: <input type="text" v-validate:address="['email']"><br />
    age: <input type="text" v-validate:age="['numeric']"><br />
    site: <input type="text" v-validate:site="['url']"><br />
    <div class="errors">
      <p v-if="$validation1.username.required">required username</p>
      <p v-if="$validation1.address.email">invalid email address</p>
      <p v-if="$validation1.age.numeric">invalid age value</p>
      <p v-if="$validation1.site.url">invalid site uril format</p>
    </div>
    <validator>
</div>
```

12.14.2 错误信息

在注册自定义验证器时，我们可以指定校验未通过时的错误信息。代码示例如下：

```
// 全局注册`email`自定义验证器
Vue.validator('email', {
  message: 'invalid email address', // 校验未通过时错误信息
  check: function (val) { // define validator
    return    /^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)*)|(\".+\"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/.test(val)
  }
})

// 重置内置验证器`required`校验未通过时错误信息
Vue.validator('required', {
  message: function (field) { // 校验未通过时错误信息
    return 'required "' + field + '" field'
  },
```

```
  check: Vue.validator('required') // 调用内置验证器
})

new Vue({
  el: '#app',
  validators: {
    numeric: { // 注册`numeric`自定义验证器
      message: 'invalid numeric value',
      check: function (val) {
        return /^[-+]?[0-9]+$/.test(val)
      }
    },
    url: { // 注册`url`自定义验证器，用来验证输入的 url 是否合法
      message: function (field) {
        return 'invalid "' + field + '" url format field'
      },
      check: function (val) {
        return /^(http\:\/\/|https\:\/\/)(.{4,})$/.test(val)
      }
    }
  },
  data: {
    email: ''
  }
})
<div id="app">
  <validator name="validation1">
    username: <input type="text" v-validate:username="['required']"><br />
    email: <input type="text" v-validate:address="['email']"><br />
    age: <input type="text" v-validate:age="['numeric']"><br />
    site: <input type="text" v-validate:site="['url']"><br />
    <div class="errors">
      <validator-errors :validation="$validation1"></validator-errors>
    </div>
  <validator>
</div>
```

12.15　自定义验证时机

在默认情况下，vue-validator 会在初始化完成后根据 validator 和 v-validate 指令自动进行验

证。但是，有时候在初始化时我们并不需要验证，这时可以通过 initial 属性或者 v-validate 验证规则来关闭自动验证。代码示例如下：

```
<div id="app">
  <validator name="validation1">
    <form novalidate>
      <div class="username-field">
        <label for="username">username:</label>
        <!-- 'inital' attribute is applied the all validators of target element (e.g. required, exist) -->
        <input id="username" type="text" initial="off" v-validate:username="['required', 'exist']">
      </div>
      <div class="password-field">
        <label for="password">password:</label>
       <!-- 'initial' optional is applied with `v-validate` validator (e.g. required only) -->
        <input id="password" type="password" v-validate:passowrd="{ required: { rule: true, initial: 'off' }, minlength: 8 }">
      </div>
      <input type="submit" value="send" v-if="$validation1.valid">
    </form>
  </validator>
</div>
```

在 validator 初始化完成后，vue-validator 会在 DOM 元素的 input、blur、change 事件触发时自动验证。可以使用 detect-change 和 detect-blur 属性来关闭这些事件的自动验证。代码示例如下：

```
<div id="app">
  <validator name="validation">
    <form novalidate @submit="onSubmit">
      <h1>user registration</h1>
      <div class="username">
        <label for="username">username:</label>
        <input id="username" type="text"
          detect-change="off" detect-blur="off" v-validate:username="{
          required: { rule: true, message: 'required you name !!' }
        }" />
      </div>
      <div class="password">
```

```
      <label for="password">password:</label>
      <input id="password" type="password" v-model="password"
        detect-change="off" detect-blur="off" v-validate:password="{
        required: { rule: true, message: 'required you new password !!' },
        minlength: { rule: 8, message: 'your new password short too !!' }
      }" />
    </div>
    <div class="confirm">
      <label for="confirm">confirm password:</label>
      <input id="confirm" type="password"
        detect-change="off" detect-blur="off" v-validate:confirm="{
        required: { rule: true, message: 'required you confirm password !!' },
        confirm: { rule: password, message: 'your confirm password incorrect !!' }
      }" />
    </div>
    <div class="errors" v-if="$validation.touched">
      <validator-errors :validation="$validation"></validator-errors>
    </div>
    <input type="submit" value="register" />
  </form>
 </validator>
</div>
new Vue({
  el: '#app',
  data: {
    password: ''
  },
  validators: {
    confirm: function (val, target) {
      return val === target
    }
  },
  methods: {
    onSubmit: function (e) {
      // validate manually
      var self = this
      this.$validate(true, function () {
```

```
        if (self.$validation.invalid) {
          e.preventDefault()
        }
      })
    }
  }
})
```

12.16 异步验证

当需要在服务端验证数据或其他需要异步验证的场景时，我们可以使用 vue-validator 的异步验证功能。

12.16.1 注册异步验证器

异步验证器的注册和同步验证器的注册流程一样，唯一的区别就是验证器方法的返回值不同，返回以下两种类型的值都为异步验证器。下面我们用 setTimeout 来说明如何注册异步验证器。

- 返回值为 function，签名为 function(resolve, reject)。调用 resolve()表示校验通过，调用 reject()表示校验失败。代码示例如下：

```
// 自定义异步验证器
Vue.validator('exist', function (val) {
  return function (resolve, reject) {
    setTimeout(function () {
      if (Math.random() > .5) {
        // 校验通过
        resolve();
      } else {
        // 校验失败
        reject();
      }
    }, 500);
  };
})

new Vue({
  el: '#app'
```

```
  data: {
    user: ''
  }
})
<div id="app">
<validator name="validation">
    user: <input type="text" v-validate:user="['exist']"><br />
    <div>
      <p v-show="$validation.user.exist">the user is already registered.</p>
    </div>
  <validator>
</div>
```

- 返回值为 promise 对象。promise 被 resolve 表示校验通过，promise 被 reject 表示校验失败。代码示例如下：

```
// 自定义异步验证器
Vue.validator('exist', function (val) {
  var promise = new Promise (function (resolve, reject) {
    setTimeout(function () {
      if (Math.random() > .5) {
        // 校验通过
        resolve();
      } else {
        // 校验失败
        reject();
      }
    }, 500);
  });
  // 返回 promise 对象
  return promise;
})
new Vue({
  el: '#app'
  data: {
    user: ''
  }
})
```

```
<div id="app">
  <validator name="validation">
    user: <input type="text" v-validate:user="['exist']"><br />
    <div>
      <p v-show="$validation.user.exist">the user is already registered.</p>
    </div>
  <validator>
</div>
```

12.16.2　验证器函数 context

验证器函数 context 是绑定到 Validation 对象上的。Validation 对象提供了一些属性，这些属性在实现特定的验证器时有用。

1. vm 属性

暴露了当前验证所在的 Vue 实例，代码示例如下：

```
new Vue({
  data () { return { checking: false } },
  validators: {
    exist (val) {
      this.vm.checking = true // spinner on
      return fetch('/validations/exist', {
        // ...
      }).then((res) => { // done
        this.vm.checking = false // spinner off
        return res.json()
      }).then((json) => {
        return Promise.resolve()
      }).catch((error) => {
        return Promise.reject(error.message)
      })
    }
  }
})
```

2. el 属性

暴露了当前验证器的目标 DOM 元素。下面展示了结合 International Telephone Input jQuery 插件使用的例子。

```
new Vue({
  validators: {
    phone: function (val) {
      return $(this.el).intlTelInput('isValidNumber')
    }
  }
})
```

第 13 章

与服务端通信

Vue.js 可以构建一个完全不依赖后端服务的应用，同时也可以与服务端进行数据交互来同步界面的动态更新。Vue.js 本身并没有提供与服务端通信的接口，但是通过插件的形式实现了基于 AJAX、JSONP 等技术的服务端通信。

vue-resource 是一个通过 XMLHttpRequest 或 JSONP 技术实现异步加载服务端数据的 Vue.js 插件。该插件提供了一般的 HTTP 请求接口和 RESTful 架构请求接口，并且提供了全局方法和 Vue 组件实例方法。一般的 HTTP 请求接口按照调用的便捷程度又分为底层方法和便捷方法，便捷方法是对底层方法的封装。在 vue-resource 中我们可以全局配置。同时，它提供了数据获取各个阶段的钩子，使得我们可以对数据获取过程进行更好的控制。以下内容讲解都是基于 vue-resource 0.7.2 版本，后面介绍中就不加版本号了。

vue-resource 插件提供的公开方法总览图如图 13-1 所示。

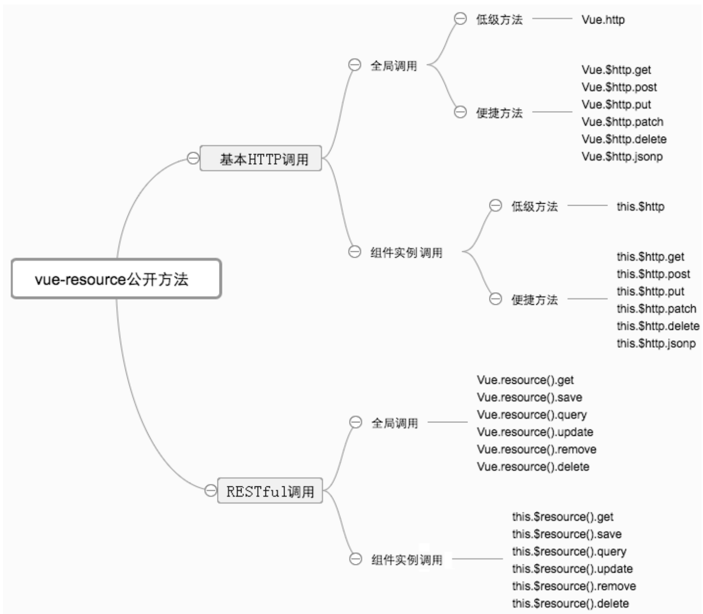


图13-1　vue-resource公开方法总览

13.1　vue-resource 安装及配置

13.1.1　安装

vue-resource 提供了 npm、bower、手动编译等安装方式，可以根据业务需要选择其中一种方式进行安装。三种安装方式的安装方法及适合场景如下：

1. npm

如果项目基于 npm 包方式来开发，则可以使用 npm 来安装 vue 和 vue-resource，执行如下命令：

```
$ npm i vue vue-resource -save-dev
```

然后在项目中引入 Vue.js 和 vue-resource，并且在 Vue.js 中注册 vue-resource 插件，代码示例如下：

```
// 引入 Vue.js 和 vue-resource
var Vue = require('vue');
var VueResource = require('vue-resource');
// 注册 vue-resource 插件
// 注意，假如 Vue.js 已经在 html 中直接引入，则不需要再执行此步骤
// 此时 vue-resource 会自动调用 Vue.use 方法来注册
Vue.use(VueResource);
```

2. bower

当业务代码使用 bower 来管理时，可以使用 bower 安装到指定目录。为了便于举例，假定该目录为 js/vendor，执行如下命令：

```
$ bower install vue-resource
```

在 HTML 中，在 vue 文件之后引入 vue-resource，代码示例如下：

```
<!-- 引入 vue -->
<script src="js/vendor/vue.js"></script>
<!-- 引入 vue-resource -->
<script src="js/vendor/vue-resource.js"></script>
```

3. 手动编译

当想尝试一些 Vue 中并未发布的新特性时，可以直接 clone 源码，手动构建来实现。由于在未正式发布之前，有些特性可能会被移除，所以不建议在生产环境中使用手动编译方式安装。

执行如下命令：

```
$ git clone https://github.com/vuejs/vue-resource.git
$ cd vue-resource
$ npm install
$ npm run build
```

编译后的文件在 dist 目录中。

13.1.2　参数配置

vue-resource 将请求配置分为全局配置、组件实例配置和调用配置三部分。这三部分的优先级依次增高，优先级高的配置会覆盖优先级低的配置。接下来我们将详细讲解每部分应如何配置。现在让我们先来看一下全局配置默认参数，如图 13-2 所示。

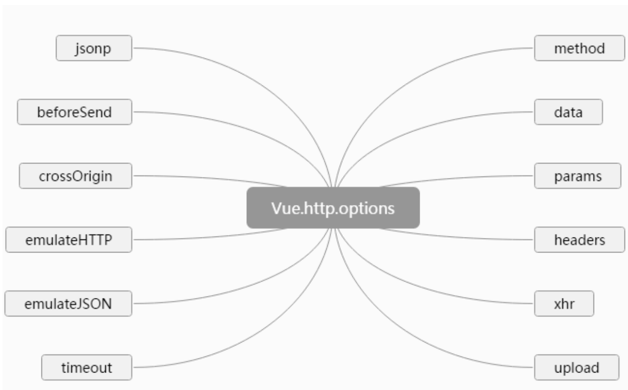


图13-2　全局配置option属性

以上参数的详细说明请参阅 13.1.5 节。

1. 全局配置

```
Vue.http.options.root = '/root'
```

2. 组件实例配置

在实例化组件时可以传入 http 选项来进行配置，代码示例如下：

```
new Vue({
  http: {
    root: '/root',
    headers: {
```

```
      Authorization: 'Basic Yxsdlfjui'
    }
  }
})
```

3. 方法调用时配置

在调用 vue-resource 请求方法时传入选项对象，代码示例如下：

```
new Vue({
  ready: function () {
    // get 请求
    this.$http.get({url: '/someUrl', headers: { Authorization: 'Basic Yxsdlfjui'}})
    .then(function (response) {
        // 请求成功回调
      }, function (response) {
        // 请求失败回调
      })
  }
})
```

13.1.3　headers 配置

在 13.1.2 节中，我们了解到可以通过 headers 属性来配置请求头。合并策略遵循参数配置合并策略。除了参数配置 headers 属性可以设置请求头外，在 vue-resource 中也提供了全局默认的 headers 配置，如图 13-3 所示。

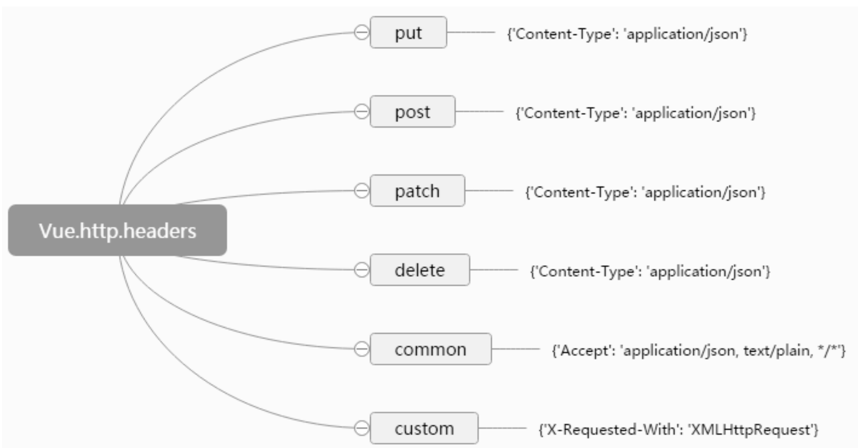


图13-3　全局默认headers配置

在图 13-3 中，我们可以看到 Vue.http.headers 键值可以是 HTTP 方法名、common、custom 三种类型。这三种类型的配置会进行合并，优先级从低到高依次是 common、custom、HTTP 方法名。

其中 common 对应的请求头会在所有请求头中设置，custom 对应的请求头在非跨域时设置，HTTP 方法名对应的请求头只有在请求的 method 匹配方法名时才会被设置。

13.1.4 基本 HTTP 调用

基本 HTTP 调用即普通的 GET、POST 等基本的 HTTP 操作，实际上执行增、删、改、查是前后端开发人员共同约定的并非通过 HTTP 的请求方法如 GET 代表获取数据、PUT 代表写入数据、POST 代表更新数据。后者为 RESTful 调用，后面我们会详细介绍 RESTful 调用方法。底层方法和便捷方法执行后返回一个 Promise 对象，可以使用 Promise 语法来注册成功、失败回调，详情请参阅 13.1.10 节。

1. 底层方法

全局的 Vue.http 方法和 Vue 组件的实例方法 this.$http 都属于底层方法，它们根据所传 option 参数的 method 属性来判断请求方式是 GET 还是 POST，抑或是其他 HTTP 的合法方法。

（1）全局调用

```
Vue.http(option)
```

（2）组件实例调用

```
this.$http(option)
```

以上两种调用方式接受相同的 option 参数，都返回 Promise 对象。不同的是，全局调用方式回调中的 this 指向 window，而组件实例调用方式回调指向组件实例。

option 对象请参阅 13.1.5 节。

我们以组件实例调用方式为例来了解一下如何发送 POST 请求，代码示例如下：

```
new Vue({
  ready: function () {
    // POST 请求
    this.$http({
      url: '/book',
      method: 'POST',
```

```
      // 请求体中发送的数据
      data: {
        cat: '1'
      },
      // 设置请求头
      headers: {
        'Content-Type': 'x-www-form-urlencoded'
      }
    }).then(function (response) {
      // 请求成功回调
    }, function (response) {
      // 请求失败回调
    })
  }
})
```

其他请求请参阅 13.1.5 节。

2. 便捷方法

不同于底层方法，便捷方法是对底层方法的封装，在调用时可以省去配置选项 option 中的 method 属性。以下为 vue-resource 提供的便捷方法列表：

- get(url, [data], [options])
- post(url, [data], [options])
- put(url, [data], [options])
- patch(url, [data], [options])
- delete(url, [data], [options])
- jsonp(url, [data], [options])

便捷方法接受三个参数：

- url（字符串），请求地址，可被 options 对象中的 url 属性覆盖。
- data（可选、字符串或对象），要发送的数据，可被 options 对象中的 data 属性覆盖。
- options，请参阅 13.1.5 节。

我们以 POST 请求为例来了解便捷方法的使用，代码示例如下：

```
this.$http.post(
  'http://example.com/book/create',
  // 请求体中要发送给服务端的数据
  {
    cat: '1',
    name: 'newbook'
  },
  {
    'headers': {
      'Content-Type': 'x-www-form-urlencoded'
    }
  }
).then(function (response) {
  // 成功回调
  console.log(response.data)
}, function (response) {
  // 失败回调
  console.log('something wrong')
})
```

13.1.5 请求选项对象

在调用 HTTP 请求方法时，可以传入选项对象来控制请求。例如：

```
Vue.http(option)
```

下面我们来看看 option 对象的各属性及含义。

1. url（字符串）

请求的 URL 地址。

2. method（字符串）

默认值为 GET，请求的 HTTP 方法（GET、POST 等）。

3. data（对象或字符串）

默认值为''，需要发送给服务端的数据。注意，data 属性的值对于 method 为 POST、PUT、DELETE 等请求会作为请求体来传送，对于 GET、JSONP 等方式的请求将会拼接在 url 查询参

数中。

4. params（对象）

默认值为{}，用来替换 url 中的模板变量，模板变量中未匹配到的属性添加在 URL 地址后边作为查询参数。代码示例如下：

```
Vue.http({
  url: 'http://example.com/{book}',
  params: {
    book: 'vue',
    cat: '1'
  }
})
```

最终 URL 为 http://example.com/vue?cat=1。

5. headers（对象）

默认值为{}，设置 HTTP 请求头。

6. xhr（对象）

默认值为 null，该对象中的属性都会应用到原生 xhr 实例对象上。源码定义如下：

```
if (_.isPlainObject(request.xhr)) {
  _.extend(xhr, request.xhr);
}
```

7. upload（对象）

默认值为 null，该对象的属性都会应用到原生 xhr 实例对象的 upload 属性上。源码实现如下：

```
if (_.isPlainObject(request.upload)) {
  _.extend(xhr.upload, request.upload);
}
```

8. jsonp（字符串）

默认值为 callback，JSONP 请求中回调函数的名字。代码示例如下：

```
Vue.http({
  url: 'http://example.com/book',
  method: 'JSONP',
  jsonp: 'cb'
})
```

最终 URL 地址为 http://example.com/book?cb=xxx，其中 xxx 为 vue-resource 生成的随机串。源码实现如下：

```
var callback = '_jsonp' + Math.random().toString(36).substr(2);
request.params[request.jsonp] = callback;
```

9. timeout（数值）

默认值为 0，单位为 ms，表示请求超时时间，0 表示没有超时限制。超时后，将会取消当前请求。vue-resource 内部通过拦截器注入超时取消逻辑。源码定义如下：

```
if (request.timeout) {
  timeout = setTimeout(function () {
    request.cancel();
  }, request.timeout);
}
//超时后 Promise 会被 reject，错误回调会被执行
```

10. beforeSend（函数）

默认值为 null，该函数接受请求选项对象作为参数。该函数在发送请求之前执行，vue-resource 内部在拦截器的最前端调用该方法。源码定义如下：

```
if (_.isFunction(request.beforeSend)) {
  request.beforeSend.call(this, request);
}
```

11. emulateHTTP（布尔值）

默认值为 false，当值为 true 时，用 HTTP 的 POST 方法发送 PUT、PATCH、DELETE 等请求，并设置请求头字段 HTTP-Method-Override 为原始请求方法。源码定义如下：

```
if (request.emulateHTTP && /^(PUT|PATCH|DELETE)$/i.test(request.method)) {
  request.headers['X-HTTP-Method-Override'] = request.method;
  request.method = 'POST';
}
```

12. emulateJSON（布尔值）

默认值为 false，当值为 true 并且 data 为对象时，设置请求头 Content-Type 的值为 application/x-www-form-urlencoded。源码定义如下：

```
if (request.emulateJSON && _.isPlainObject(request.data)) {
  request.headers['Content-Type'] = 'application/x-www-form-urlencoded';
  request.data = _.url.params(request.data);
}
```

13. crossOrigin（布尔值）

默认值为 null，表示是否跨域，如果没有设置该属性，vue-resource 内部会判断浏览器当前 URL 和请求 URL 是否跨域。源码定义如下：

```
if (request.crossOrigin === null) {
  request.crossOrigin = crossOrigin(request);
}
if (request.crossOrigin) {
  if (!xhrCors) {
    request.client = xdrClient;
  }
  request.emulateHTTP = false;
}
```

如果最终 crossOrigin 为 true 并且浏览器不支持 CORS，即不支持 XMLHttpRequest2 时，则会使用 XDomainRequest 来请求。目前 XDomainRequest 只有 IE 8、IE 9 两款浏览器支持用来进行 AJAX 跨域。XMLHttpRequest2 和 XdomainRequest 的具体细节请参阅 13.1.9 节。

13.1.6　response 对象

response 对象包含服务端返回的数据，以及 HTTP 响应状态、响应头等信息。下面展示了 response 对象的各属性及其含义。

- data（对象或字符串）：服务端返回的数据，已使用 JSON.parse 解析。
- ok（布尔值）：当 HTTP 响应状态码在 200～299 区间时该值为 true，表示响应成功。
- status（数值）：HTTP 响应状态码。
- statusText（字符串）：HTTP 响应状态文本描述。
- headers（函数）：获取 HTTP 响应头信息，不传参表示获取整个响应头，返回一个响应头对象；传参表示获取对应的响应头信息。
- request（对象）：请参阅 13.1.5 节。

13.1.7　RESTful 调用

RESTful 调用方式就是客户端通过 HTTP 动词来表示增、删、改、查实现对服务端数据操

作的一种架构模式。

vue-resource 提供全局调用 Vue.resource 或者在组件实例上调用 this.$resource。这两种调用方式接受相同的参数：

```
resource(url, [params], [actions], [options])
```

1. url（字符串）

请求地址，可以包含占位符（花括号所包围的内容即为占位符），它会被 params 对象中的同名属性的值替换，详情请参阅 13.1.11 节。

```
this.$resource('/books/{cat}', {cat: '1'})
```

最终实际 URL 为/books/1。

2. params（可选，对象）

参数对象，可用来替换 url 中的占位符，多出来的属性会拼接成 url 的查询参数。

3. actions（可选，对象）

可以用来对已有的 action 进行配置，也可以用来定义新的 action。默认的 action 配置如下：

```
Resource.actions = {
  get: {method: 'GET'},
  save: {method: 'POST'},
  query: {method: 'GET'},
  update: {method: 'PUT'},
  remove: {method: 'DELETE'},
  delete: {method: 'DELETE'}
}
```

我们可以定义新的 action，代码示例如下：

```
this.$resource(
  '/books/{cat}',
  {
    cat: '1'
  },
  {
    charge:
    {
      method: 'POST',
      params: {
```

```
        charge: true
      }
    }
  }
)
```

上面我们定义了一个名为 charge 的 action，它使用 POST 方法来请求资源。

注：actions 对象中的单个 action 如 charge 对象可以包含 options 中的所有属性，且其优先级高于 options 对象。

4. options（可选，对象）

请参阅 13.1.5 节。

resource 方法执行后返回一个包含了所有 action 方法名的对象，其包含自定义的 action 方法，这些方法都返回 Promise 对象，详情请参阅 13.1.10 节。下面让我们看看完整地使用 resource 来请求数据的例子。代码示例如下：

```
var resource = this.$resource('/books/{id}');
// 查询
// 第一个参数为 params 对象，优先级高于 resource 方法的 params 参数
resource.get({id: 1}).then(response) {
  this.$set('item', response.item)
}
// 保存
// 第二个参数为要发送的数据
resource.save({id: 1}, {item: this.item}).then(function (response) {
  // 请求成功回调
}, function (response) {
  // 请求失败回调
})
resource.delete({id: 1}).then(function (response) {
  // 成功回调
}, function (response) {
  // 失败回调
})
```

13.1.8　拦截器

可以全局进行拦截器设置，拦截器可以在请求发送前或响应返回时做一些特殊的处理。

1. 拦截器的注册

```
Vue.http.interceptors.push({
  request: function (request) {
    // 更改请求类型为 POST
    request.method = 'POST'
    return request
  },
  response: function (response) {
    // 修改返回数据
    response.data = [{
      custom: 'custom'
    }]
    return response
  }
})
```

2. 工厂函数注册

```
Vue.http.interceptors.push(function () {
  return {
    request: function (request) {
      return request
    },
    response: function (response) {
      return response
    }
  }
})
```

13.1.9　跨域 AJAX

很多人认为 AJAX 只能在同域的情况下发送成功，很早的时候，由于浏览器安全策略，AJAX 确实只能在同域的情况下发送。但是目前很多浏览器已经开始支持 XMLHttpRequest2。XMLHttpRequest2 引入了大量的新特性，例如跨域资源请求（CORS）、上传进度事件、支持二进制数据上传/下载等。

本节我们将会介绍 vue-resource 中用到的 CORS 特性，以及 XMLHttpRequest2 的替代品 XDomainRequest。

1. XMLHttpRequest2 CORS

XMLHttpRequest2 是第二代 XMLHttpRequest 技术，提交 AJAX 请求还是和普通的 XMLHttpRequest 请求一样，只是增加了一些新特性。

在提交 AJAX 跨域请求时，首先我们需要知道当前浏览器是否支持 XMLHttpRequest2，判断方法是使用 in 操作符检测当前 XMLHttpRequest 实例对象是否包含 withCredentials 属性，如果包含则支持 CORS。代码示例如下：

```
var xhrCors = 'withCredentials' in new XMLHttpRequest()
```

在支持 CORS 的情况下，还需要服务端启用 CORS 支持。

假如我们想从 http://example.com 域中提交请求到 http://crossdomain.com 域，那么需要在 crossdomain.com 域中添加如下响应头：

```
Access-Control-Allow-Origin: http://example.com
```

如果 crossdomain.com 要允许所有异域都可以 AJAX 请求该域资源，则添加如下响应头：

```
Access-Control-Allow-Origin: *
```

服务端开启 CORS 支持后，在浏览器中我们就可以和提交普通的 AJAX 请求一样提交跨域请求了。代码示例如下：

```
var xhr = new XMLHttpRequest()
xhr.open('GET', 'http://www.crossdomain.com/hello.json')
xhr.onload = function(e) {
  var data = JSON.parse(this.response)
  ...
}
xhr.send()
```

2. XDomainRequest

如果想在 IE 8、IE 9 中支持 CORS（IE 10 及以后版本支持 XMLHttpRequest2），我们可以使用 XdomainRequest（该属性目前已废弃，不建议使用）。如果 vue-resource 不支持 XMLHttpRequest2，则会降级使用此种方式。代码示例如下：

```
// 实例化 XDomainRequest
var xdr = new XDomainRequest()
xdr.open("get", "http://crossdomain.com/hello.json")
xdr.onprogress = function () {
  // 进度回调
```

```
}
xdr.ontimeout = function () {
  // 超时回调
}
xdr.onerror = function () {
  // 出错回调
}
xdr.onload = function() {
  // 成功回调
  //success(xdr.responseText)
}
setTimeout(function () {
  // 发送请求
  xdr.send()
}, 0)
```

注：XDomain 只支持 GET 和 POST 两种请求，如果要在 vue-resource 中使用其他方法请求，请设置请求选项对象的 emulateHTTP 为 true。在定时器中调用 xhr.send()方法，是为了防止多个 XDomainRequest 请求同时发送时部分请求丢失。

13.1.10　Promise

vue-resource 基本 HTTP 调用和 RESTful 调用 action 方法执行后都会返回一个 Promise 对象，该 Promise 对象提供了 then、catch、finally 等常用方法来注册回调函数。代码示例如下：

```
var promise = this.$http.post(
  'http://example.com/book/create',
  // 请求体中要发送给服务端的数据
  {
    cat: '1',
    name: 'newbook'
  },
  {
    'headers': {
      'Content-Type': 'x-www-form-urlencoded'
    }
  }
)
promise.then(function (response) {
```

```
  // 成功回调
  console.log(response.data)
}, function (response) {
  // 失败回调
  console.log('something wrong')
})
promise.catch(function (response) {
  // 失败回调
  console.log('something wrong')
})
promise.finally(function () {
  // 执行完成功或者失败回调后都会执行此逻辑
})
```

注：所有回调函数的 this 都指向组件实例。

13.1.11　url 模板

vue-resource 使用 url-template 库来解析 url 模板。我们来看一个使用 url-template 解析 url 模板的例子。代码示例如下：

```
// 引入 url-template 库
var template = require('url-template');
var emailUrl = template.parse('/{email}/{folder}/{id}');
// 返回 '/user@domain/test/42'
emailUrl.expand({
  email: 'user@domain',
  folder: 'test',
  id: 42
})
```

在 vue-resource 方法请求传参时我们可以在 url 中放置花括号包围的占位符，vue-resource 内部会使用 url-template 将占位符用 params 对象中的属性进行替换。代码示例如下：

```
this.$resource('/books/{cat}', {cat: '1'})
```

最终实际 URL 为/books/1。

注：url-template：https://github.com/bramstein/url-template。

13.2　vue-async-data

vue-async-data 是一个异步加载数据状态指示的插件，它本身并不支持异步获取服务端数据的功能，仅仅指示数据目前是处于加载状态还是已经加载完毕。通过不同的状态我们可以设置加载动画效果等。

13.2.1　安装

如果项目基于 npm 包方式来开发，则可以使用 npm 来安装 vue 和 vue-async-data。执行如下命令：

```
$ npm i vue vue-async-data -save-dev
```

然后在项目中引入 Vue.js 和 vue-async-data，并且在 Vue.js 中注册 vue-async-data 插件。代码示例如下：

```
// 引入 Vue.js 和 vue-async-data
var Vue = require('vue')
var VueAsyncData = require('vue-async-data')
// 注册 vue-async-data 插件
Vue.use(VueAsyncData)
```

13.2.2　使用

在创建 Vue 组件实例时，在选项中增加 asyncData 方法。代码示例如下：

```
// 假设为 CommonJS 环境
var Vue = require('vue')
var VueAsyncData = require('vue-async-data')
// 安装 vue-async-data 插件
Vue.use(VueAsyncData)
// 在创建组件实例时，在选项中增加 asyncData 方法
Vue.component('example', {
  data: function {
    return {
      msg: 'not loaded yet...'
    }
  },
  asyncData: function (resolve, reject) {
```

```
    // 数据加载成功时调用 resolve(data)，告诉 vue-async-data 数据加载成功，此时$loadingAsyncData
    // 会被设置为 false
    // 数据加载失败时调用 reject(reason)，告诉 vue-async-data 数据加载失败
    // 以下为了便于举例，我们用 setTimeout 来模拟异步数据请求，实际上可以调用任何数据请求插件，如
    // vue-resource
    setTimeout(function () {
      // 以下方法调用后，vue-router 会自动调用 vm.$set('msg','hi')，vm.$set('$loadingAsyncData',true)
      resolve({
        msg: 'hi'
      })
    }, 1000)
  }
})
```

假设上面的 example 组件对应的视图模板如下：

```
<div v-if="$loadingAsyncData">Loading...</div>
<div v-if="!$loadingAsyncData">Loaded. Put your real content here.</div>
```

那么，当组件创建时会展示 Loading…内容；当数据成功返回时会展示 Loaded. Put your real content here.内容。

有时候我们需要手动去服务端获取最新数据来更新界面，这时需要调用组件的实例方法来重新加载数据。代码示例如下：

```
vm.reloadAsyncData()
```

注：必须调用 vm.reloadAsyncData()，而不是 vm.asyncData()。因为在 vue-router 内部实现时，在调用 vm.reloadAsyncData()方法时首先会将 vm.$loadingAsyncData 设置为 true，在数据成功返回时会设置 vm.$loadingAsyncData 为 false。这样模板就会根据$loadingAsyncData 的真假来判断该展示 Loading 效果还是实际内容。

13.3　常见问题解析

13.3.1　如何发送 JSONP 请求

首先我们需要知道 JSONP 是利用 JavaScript 可以跨域的特性从服务端请求数据的。也就是说，在跨域的情况下才有必要使用 JSONP 来发送请求。vue-resource 提供了三种调用方式。

1. 全局方法

```
Vue.http({
  url: 'http://example.com/books',
  //参数部分，将会拼接在 url 之后
  params: {
    cat: 1
  },
  method: 'JSONP'
}).then(function (response) {
  // response.data 为服务端返回的数据
  console.log(response.data)
}).catch(function (response) {
  // 出错处理
  console.log(response)
})
```

2. 实例底层方法

```
this.$http({
  url: 'http://example.com/books',
  // 参数部分，将会拼接在 url 之后
  params: {
    cat: 1
  },
  method: 'JSONP'
}).then(function (response) {
  // this 指向当前组件实例
  console.log(this)
}).catch(function (response) {
  // 出错处理
  console.log(response)
})
```

3. 实例便捷方法

```
this.$http.jsonp(
  'http://example.com/books',
  // 参数部分，将会拼接在 url 之后
  {
    cat: 1
  }
).then(function (response) {
```

```
  // this 指向当前组件实例
  console.log(this)
}).catch(function (response) {
  // 出错处理
  console.log(response)
})
```

13.3.2　如何修改发送给服务端的数据类型

在默认情况下，对于 PUT、POST、PATCH、DELETE 等请求，请求头中的 Content-Type 为 application/json，即 JSON 类型。有时我们需要将数据提交为指定类型，如 application/x-www-form-urlencoded、multipart/form-data、text/plain 等。下面我们以 POST 请求为例来说明。

1. 全局 headers 配置

```
Vue.http.headers.post['Content-Type'] = 'application/x-www-form-urlencoded'
```

2. 实例配置

```
this.$http.post(
  'http://example.com/books',
  // 成功回调
  function(data, status, request) {
    if (status == 200) {
      console.dir(data)
    }
  },
  {
    // 配置请求头
    headers: {
      'Content-Type': 'multipart/form-data'
    }
  }
)
```

注：实例配置的优先级高于全局配置，因此最终 Content-Type 为 multipart/form-data。

13.3.3　跨域请求出错

跨域请求需要服务端开启 CORS 支持，详情请参阅 13.1.9 节。

13.3.4 $.http.post 方法变为 OPTIONS 方法

在跨域的情况下，对于非简单请求（PUT、DELETE、Content-Type 为 application/json），浏览器会在真实请求前额外发起一次类型为 OPTIONS 的请求。只有服务器正确响应了 OPTIONS 请求后，浏览器才会发起真实请求。

因此，为了在跨域的情况下使用 POST 提交 Content-Type 为 application/json 的数据或 PUT、DELETE 等非简单请求，首先服务端需要开启 CORS 支持，详情请参阅 13.1.9 节。

同时需要设置如下响应头：

```
Access-Control-Allow-Methods: POST, GET, PUT, DELETE, OPTIONS
```

第 14 章

路由与视图

同 AngularJS 一样，Vue.js 也很适合用来做大型单页应用。Vue.js 本身并没有提供路由机制，但是官方以插件（vue-router）的形式提供了对路由的支持。vue-router 0.7.13 支持嵌套路由、组件惰性载入、视图切换动画、具名路径等特性。以下内容讲解都是基于 vue-router 0.7.13 版本，后面就不加版本号了。

14.1 如何安装

vue-router 提供了 npm、bower、手动编译等安装方式，可以根据业务需要选择其中一种方式进行安装。这三种安装方式的安装方法及适合场景如下：

1. npm

当业务代码使用 Webpack 等支持 CommonJS 规范的模块化打包器来构建时，可以使用 npm 包的方式来安装：

```
$ npm install vue-ruoter
```

因为 vue-router 是 Vue.js 的一个插件，所以 vue-router 需要使用 Vue.use(PluginContructor)（Vue.js 用此方法来注册插件）注册到 Vue 对象上，在 vue-router 内部会检测 window.Vue 对象是否存在，如果存在则会自动调用 Vue.use()方法，否则需要使用者手动调用 Vue.use(VueRouter)来确保路由插件注册到 Vue 中。在 Webpack 等支持 CommonJS 规范的环境中，Vue 对象并不会暴露到全局 window 对象中，而是会通过 module.exports 形式输出，因此需要使用者手动注册。

```
var Vue = require('vue')
var VueRouter = require('vue-router')
// 安装 vue-router
Vue.use(VueRouter)
```

2. bower

当业务代码使用 bower 来管理时，可以使用 bower 安装到指定目录。为了便于举例，假定该目录为 js/vendor。

```
$ bower install vue-router
```

在 HTML 中，在 vue 文件之后引入 vue-router：

```
<!-- 引入 vue -->
<script src="js/vendor/vue.js"></script>
<!-- 引入 vue-router -->
<script src="js/vendor/vue-router.js"></script>
```

3. 手动编译

当想尝试一些 Vue 中并未发布的新特性时，可以直接 clone 源码，手动构建来实现。由于在正式发布之前，有些特性可能会被移除，所以不建议在生产环境中使用手动编译方式安装。

```
$ git clone https://github.com/vuejs/vue-router.git
$ cd vue-router
$ npm install
$ npm run build
```

14.2 基本使用

为了快速入门，我们先来看一个基于 Webpack 构建的简单示例。代码示例如下：

```
<div id="app">
  <h1>Hello App!</h1>
  <p>
    <!-- 使用指令 v-link 进行导航 -->
    <a v-link="{ path: '/foo' }">Go to Foo</a>
    <a v-link="{ path: '/bar' }">Go to Bar</a>
  </p>
  <!-- 路由外链 -->
  <router-view></router-view>
</div>

// 引入 Vue.js
var Vue = require('vue');
// 引入 vue-router
var VueRouter = require('vue-router');
```

```
// 定义各路由组件
var Foo = Vue.extend({
  template: '<p>This is foo!</p>'
})
var Bar = Vue.extend({
  template: '<p>This is bar!</p>'
})
// 路由器需要一个根组件
// 出于演示的目的，这里使用一个空的组件，直接使用 HTML 作为应用的模板
var App = Vue.extend({})
// 创建一个路由器实例
// 创建实例时可以传入配置参数进行定制。为保持简单，这里使用默认配置
var router = new VueRouter()
// 定义路由规则
// 每条路由规则应该映射到一个组件。这里的“组件”可以是一个使用 Vue.extend
// 创建的组件构造函数，也可以是一个组件选项对象
// 稍后我们会讲解嵌套路由
router.map({
  '/foo': {
    component: Foo
  },
  '/bar': {
    component: Bar
  }
})
// 现在我们可以启动应用了
// 路由器会创建一个 App 实例，并且挂载到选择符 #app 匹配的元素上
router.start(App, '#app')
```

14.3　视图部分

视图部分用来给用户提供导航以及导航结果展示区域。

14.3.1　v-link

在原生 HTML 中，我们用<a>标签的 href 属性来导航。在 vue-router 应用中，我们还是使用<a>标签，不同的是，我们使用 v-link 属性而不是 href 属性。代码示例如下：

```
<a v-link="{ path: '/join/DDFE' }">Join DDFE</a>
```

当用户在页面上点击 Join DDFE 时，vue-router 会在路由映射中匹配 path 为/join/DDFE 的路由规则，如果成功匹配到，则会将对应路由组件的模板内容渲染到 router-view 区域中。

v-link 是一个 Vue.js 指令，它的值是一个 JavaScript 表达式，可以接受一个表示 path 的字符串或者包含 name 或 path 属性的对象。如果属性值既不是字符串也不是对象字面量，则会被当作对应组件的数据属性来解析。代码示例如下：

```
<!-- 字面量路径，表示 path。注意：需要用单引号把字符串内容括起来 -->
<a v-link="'/home'">Home</a>

<!-- 值为对应组件的数据属性，效果同上 -->
Vue.component('app', {
  data: {
    homeLinkMap: {
      path: '/home'
    }
  }
})

<a v-link="homeLinkMap">Home</a>
<!-- 值是一个包含 path 属性的对象，效果同上 -->
<a v-link="{ path: '/home' }">Home</a>
<!-- 值是一个包含 name 属性的对象 -->
<a v-link="{ name: 'order', params: { status: 0}}">order</a>
```

当 v-link 解析后的值是对象时，该对象可以有以下属性。

1. params（对象）

包含路由中的动态片段和全匹配片段的键值对。动态片段和全匹配片段请参阅 14.6 节。

2. query（对象）

包含路由中添加到路径 path 后的键值对。代码示例如下：

```
<a v-link="{ path: '/home', query: {isAuthed: true} }">Home</a>
```

当该 path 被匹配时，地址栏 URL 为：/home?isAuthed=true。

3. replace（布尔值）

默认值为 false。当该值为 true 时，此次导航不会产生历史记录。

4. append（布尔值）

默认值为 false。当该值为 true 时，如果此次导航的目的 path 为相对路径，则实际 URL 中的路径是当前 path 后拼接目的 path。假设当前 path 为/a，代码示例如下：

```
<!-- 不加 append 属性（默认值为 false），目的 URL 路径为/b -->
<a v-link="{ path: 'b' }">b</a>

<!-- append 为 true，目的 URL 路径为/a/b -->
<a v-link="{ path: 'b', append: true}"> /a/b </a>
```

5. activeClass（字符串）

默认值为 v-link-active，指带有 v-link 指令的 a 元素处于激活状态时的 class 名称。该值也可以在创建路由器实例时通过选项的 linkActiveClass 属性来进行全局设置。

注：在判断当前 v-link 指令所在元素是否处于激活状态时默认使用的是包含匹配。也就是说，当前实际匹配的路由 path 中完全包含 v-link 所指 path 时该元素处于激活状态。注意是实际匹配路由中的 path，而不是浏览器地址栏 hash 中的 path 部分，强调这个是因为当我们配置路由别名时，地址栏的 path 和实际路由的 path 并不一致。我们也可以通过 exact:true 来设置只有当 v-link 中的 path 和实际路由的 path 完全相等时才算匹配。代码示例如下：

```
<a v-link="{path: '/exact', exact: true}">exact active</a>
```

注：使用 v-link 而不是 href 来设置 URL，原因如下。

- v-link 是一个 Vue.js 指令，它会根据它的值来设置 href 的值。
- 在 hash 模式和 HTML5 history 模式下，vue-router 会统一行为，这样在改变模式时不需要做任何改变。
- 在 HTML5 history 模式下，v-link 指令会监听点击事件，防止浏览器重新加载页面。
- 在 HTML5 history 模式下，如果使用 root 选项，不需要在 v-link 的 path 中包含 root 路径。
- 在 Vue.js 1.0 绑定语法中，不支持 Mustache 插值标签，可以使用常规的 JavaScript 表达式代替，例如 v-link = "'user/' + user.name"。

14.3.2 router–view

视图部分用来展示匹配路由的模板内容，在 vue-router 中使用 router-view 来渲染匹配的组件。router-view 是一个 Vue 组件，它具有以下一些特性。

- 通过 props 传递数据。
- 支持 v-transition 和 transition-mode，代码示例如下：

```
<!-- 可以使用 transition 指令在路由切换时提供过渡效果 -->
<router-view transition="demo" transition-mode="out-in"></router-view>
```

- 支持 v-ref，被渲染的组件会注册到父级组件的 this.$对象中。
- 支持 slot，router-view 中的 HTML 内容会被插入到相应路由组件模板的 slot 中。

14.4 路由实例

在开始使用 vue-router 开发路由应用时，首先我们需要实例化 vue-router。代码示例如下：

```
var VueRouter = require('vue-router')
var router = new VueRouter(vueRouterConfig)
```

实例化 VueRouter 时可以传入一个可选的 vueRouterConfig 路由选项对象来自定义路由器的行为。返回 router 路由器实例，router 实例暴露了一些实例属性和实例方法，我们可以用来控制整个路由应用。接下来介绍路由选项对象和路由器实例属性、方法。

1. 路由选项

创建路由器实例时，可以传入路由选项来自定义路由器行为。

可选参数如下：

（1）hashbang（布尔值）

默认值为 true。当该值为 true 时，表示匹配的路由在浏览器地址栏中以 hash 模式显示。例如：假设当前浏览器地址栏中的地址为 http://example.com/path?query，当用户点击 home 链接时，浏览器地址栏中的地址会显示为 http://example.com/path?query#!/home。

```
<a v-link="{path: '/ddfe'}">Join DDFE</a>
```

（2）history（布尔值）

默认值为 false。当该值为 true 时，会以 HTML5 history API 进行导航。当 history 值为 true 时，需要注意以下问题：

- 假如当前页面地址为 http://example.com/home，而在路由配置中配置了/home/welcome 路径，那么当用户直接访问 http://example.com/home/welcome 路径时，服务器端应确保返回 http://example.com/home 页面，而不是 http://example.com/home/welcome 页面；否则有可能因为/home/welcome 页面不存在而返回 404 错误。
- 当 history 值为 true 时，不论 hashbang 值是否为 true，总会以 history 模式进行导航。
- 当 history 值为 true，而浏览器并不支持 HTML5 history API 时，vue-router 会自动降级为 hashbang 模式。

（3）saveScrollPosition（布尔值）

默认值为 false，该值只在 history 值为 true 时生效。当该值设置为 true 时，在点击浏览器后退按钮时页面会定位到上一次该路由对应视图所在位置。

（4）transitionOnLoad（布尔值）

默认值为 false。当该值为 true 时，在页面第一次加载时 router-view 会有路由切换动画，默认为直接渲染。

（5）suppressTransitionError（布尔值）

默认值为 false。当该值为 true 时，在组件路由切换钩子中产生的异常不会被抛出。

（6）linkActiveClass（字符串）

默认值为 v-link-active，表示 v-link 所在元素处于激活状态时 vue-router 加在该元素上的类名。

（7）root（字符串）

默认值为 null，该值只在 history 值为 true 时生效。定义路由根路径，所有路径被匹配时，浏览器地址栏 URL 会显示为根路径+匹配路径。

2. 路由器实例属性

router 实例暴露了两个属性：

（1）app（根组件实例）

vue-router 应用的根 Vue 实例，由调用 router.start(App,'#app')时传入的组件构造器 App 创建得到。

（2）mode（字符串）

可能值有 html5、hash、abstract。

- html5：当创建 router 实例时，所传配置对象 history 值为 true，并且浏览器支持 HTML5 history API 时。
- hash：当创建 router 实例时，所传配置对象 hash 值为 true，或者 history 值为 true，但浏览器不支持 HTML5 history API 时。
- abstract：当宿主环境中没有 window 对象（例如非浏览器环境）时，会自动退化为此模式。

3. 路由器实例方法

router 实例对外暴露了很多方法，用来提供启动、路由映射、重定向、路由切换全局钩子等功能。具体的方法名及实现功能如下。

（1）start(App,el)

启动路由应用。该方法接受两个参数：

- App（函数或对象）

App 可以是一个 Vue 组件构造器或者组件选项对象，当为组件选项对象时，在 vue-router 内部会调用 Vue.extend 来创建 App 构造器。

- el（字符串或 DOM 元素）

el 可以是一个 CSS 选择器或者 DOM 元素，用来挂载路由应用的根组件。

（2）On(path,config)

添加顶级路由配置。该方法接受两个参数：

- path（字符串）

要匹配的路径，请参阅“路由匹配”部分。

❍ config（对象）

路由配置对象，请参阅“路由配置对象”部分。

（3）Map(routerMap)

批量定义路由映射规则，内部调用 router.on 方法实现。该方法接受一个参数：

❍ routerMap（对象）

参数 routerMap 为对象，键为路径，值为路由配置对象。在 vue-router 内部会对 routerMap 对象中的每个键值对调用 router.on()方法来进行路由映射。路径定义请参阅“路由匹配”部分，路由配置对象请参阅“路由配置对象”部分。

（4）go(path)

导航到指定 path 的路由。该方法接受一个参数：

❍ path（字符串或对象）

当 path 为字符串时，会当作普通路径来解析。如果路径是相对路径（不以“/”开头），则会以相对于当前路径的方式进行解析。

当 path 为对象时，对象中只包含 path 属性：

```
{path: '/a/b}
```

或者

```
{
  name: 'order',
  params: {id: 1},
  query: {fieldName: 'address'}
}
```

包含 name 的路径及具名路径，可参阅“具名路径”部分。

当 path 为对象时，两种格式都支持可选 replace 和 append 属性：

➢ replace，布尔类型，默认值为 false。当该值为 true 时，跳转不产生新的历史记录。

➢ Append，布尔类型，默认值为 false。当该值为 true 时，假如要跳转到的路径是相对路

径，则实际路径是当前路径拼接要跳转到的路径。假设当前路径为/a，目的路径为 b，当 append 值为 false 时，则实际跳转后路径为/b；为 true 时，则实际跳转后路径为/a/b。

（5）replace(path)

和 router.go(path)类似，但不会创建新的历史记录。其参数与 replace.go(path)的参数相同。

（6）redirect(redirectMap)

定义全局重定向规则。如果要访问的路径匹配重定向规则，则路径会被重定向到指定的路径，以重定向后的路径在浏览器中生成历史记录，原本访问的路径不会生成历史记录。该方法接受一个参数：

❍ redirectMap（对象）

该参数格式为{fromPath: toPath}，即当前访问的路径到实际路径的映射关系。fromPath 和 toPath 请参阅“路由匹配”部分。

（7）Alias(aliasMap)

配置别名规则 。和重定向不同的是 ，重定向的地址栏显示的是 toPath ，实际匹配的也是 toPath；而别名实际地址栏显示的是 fromPath，而路由实际匹配的是 toPath。该方法接受一个参数：

➢ aliasMap（对象）

该参数格式为{fromPath: toPath}，fromPath 和 toPath 请参阅“路由匹配”部分。

（8）BeforeEach(hookFunction)

该方法用来全局注册前置钩子。它会在整个路由切换的最前端被调用，优先于各路由组件中的路由钩子执行，因此这里可以做一些全局的访问控制。当钩子被 reject 时，整个路由切换将取消，页面将保持原来的状态。

我们可以用该方法来注册多个钩子函数，这些钩子会按照注册的顺序调用。但是，需要注意的是，后一个钩子只有在前一个钩子被 resolve 之后才会调用。

❍ hookFunction（函数）

钩子函数，它接受一个 transition 对象作为参数。关于 transition 对象请参阅“transition 对象”部分。

（9）router.afterEach(hookFunction)

该方法用来全局注册后置钩子函数，该钩子会在每次 canDeactivate 和 canActivate 钩子被 resolve 之后执行，并不能保证所有的 activate 钩子被 resolve。

我们可以用该方法注册多个全局后置钩子，这些钩子会按照注册顺序调用。但和全局前置钩子不同的是，后一个钩子并不会等前一个钩子执行完才执行，它们是并行执行的。

- hookFunction（函数）

该函数接受一个 transition 对象作为参数，但只能从该对象访问 from 和 to 属性，这两个属性分别是当前和将要切换到的路由对象。在后置钩子中，transition 对象没有提供任何控制路由流程的方法。关于 transition 对象请参阅“transition 对象”部分。

14.5　组件路由配置

在 vue-router 应用中，每一个路由对应一个组件，在路由组件中我们可以配置 route 字段来实现在路由切换的各个阶段对组件进行更好的控制：

```
var Message = Vue.extend({
  route: {
    canActivate: function (transition) {
      transition.next()
    }
  }
})
```

以上示例中，route 对象中的 canActivate 方法用来在组件即将切入之前判断是否可以激活，只有可以激活才会成功切入。在 route 对象中还可以配置其他阶段的钩子方法，以便更好地控制切换流程。

14.5.1　路由切换的各个阶段

vue-router 将路由切换分为三个阶段。为了更好地理解各个阶段完成的工作，我们先假定当前匹配的路径为 /a/b/c，此路径对应三个嵌套的 router-view，如图 14-1 所示。

如图 14-2 所示，当用户接下来要访问的路径是 a/d/e 时，我们需要从 a/b/c 路径对应的组件树切换到 a/d/e 对应的组件树。

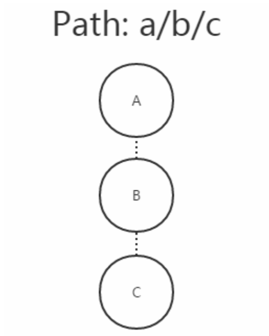


图14-1　路径a/b/c对应的嵌套组件树

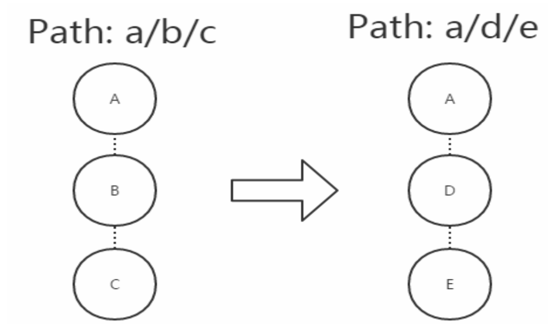


图14-2　a/b/c和a/d/e对应的组件树

在以上路由切换过程中，我们需要做以下工作：

- 可以重用组件 A，因为重新渲染后，组件 A 依然保持不变。
- 需要停用并移除组件 B 和 C。
- 启用并激活组件 D 和 E。
- 在执行步骤 2 和 3 之前，需要确保切换效果有效，也就是确保切换中涉及的所有组件都能按照所期望的那样被停用/激活。

使用 vue-router，我们可以通过实现切换钩子函数来控制这些步骤。在讲解如何做的细节之前，我们先来了解一下大局。

路由切换的各个阶段如下：

1. 可重用阶段

如图 14-3 所示，检查当前视图结构中是否存在可以重用的组件。这是通过对比两棵新的组件树，找出共用的组件，然后检查它们的可重用性（通过 canReuse 选项）来判断的。在默认情况下，所有组件都是可重用的，除非是定制过的。

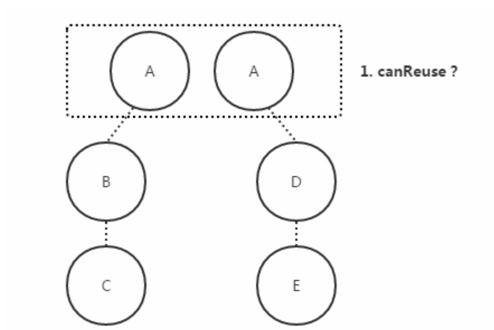


图14-3　可重用阶段

2. 验证阶段

如图 14-4 所示，检查当前组件是否能够停用，以及新组件是否可以被激活。这是通过调用路由配置阶段的 canDeactivate 和 canActivate 钩子函

数来判断的。

注：canDeactivate 按照从下至上的冒泡顺序检查，而 canActivate 则是从上至下。

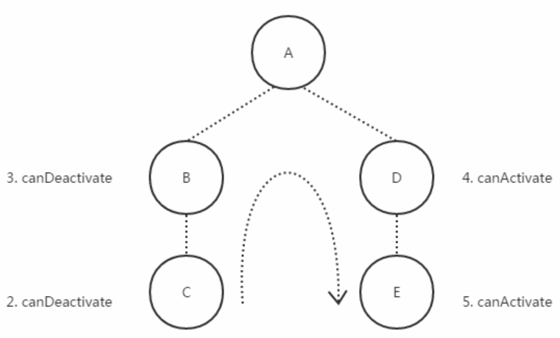


图14-4　验证阶段

任何一个钩子函数都可以终止界面切换。如果在验证阶段终止了界面切换，路由器会保持当前应用状态，恢复到前一个路径。

3. 激活阶段

如图 14-5 所示，一旦所有的验证钩子函数都被调用而且没有终止切换，切换就可以认定是合法的，路由器则开始禁用当前组件并启用新组件。

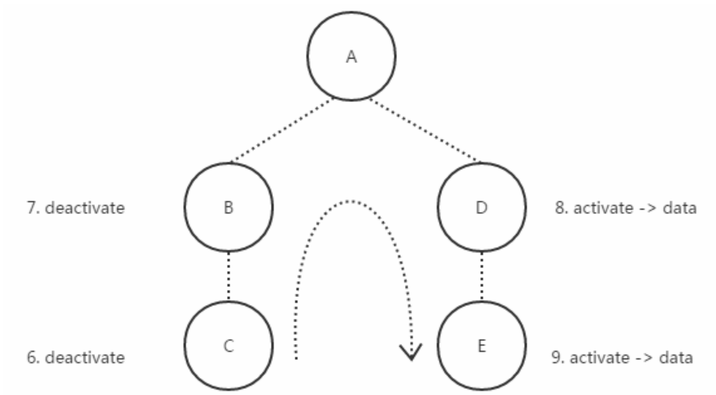


图14-5　激活阶段

此阶段对应钩子函数的调用顺序和验证阶段相同，其目的是在组件切换真正执行之前提供一个进行清理和准备的机会。界面的更新会等到所有受影响组件的 deactivate 和 activate 钩子函数执行之后才进行。

data 这个钩子函数会在 activate 之后被调用，或者在当前组件可以重用时也会被调用。

接下来介绍一下在切换过程中各个钩子函数的细节。

14.5.2 各阶段的钩子函数介绍

在路由切换的各个阶段，vue-router 都提供了相应的钩子函数，这些钩子函数包括：

- canReuse
- canActivate
- activate
- data
- canDeactivate
- deactivate

可以在实例化组件时在 route 选项中实现这些钩子函数，代码示例如下：

```
var Message = Vue.extend({
  route: {
    canActivate: function (transition) {
      transition.next()
    }
  }
})
```

每个钩子函数都会接受一个 transition 对象作为参数，详见 14.7 节。

1. 钩子函数异步 resolve

通常我们会在钩子函数中进行一些异步操作，在异步操作被 resolve 之前，切换会处于暂停状态。钩子函数 resolve 遵循以下规则：

- 如果钩子函数返回一个 Promise 对象，则钩子函数何时 resolve 取决于该 Promise 何时 resolve。代码示例如下：

```
route {
  activate: function () {
    // 返回 Promise 对象
```

```
    return new Promise(function (resolve, reject) {
      if (Math.random() > 0.5) {
        // resolved
        resolve('resolved')
      } else {
        // rejected
        reject('rejected')
      }
    })
  }
}
```

- 如果钩子函数既不返回 Promise 对象，也没有任何参数，则该钩子函数将被同步 resolve。代码示例如下：

```
route: {
  activate: function (/* 没有参数 */) {
    // 如果不返回 Promise 对象，则同步 resolve
  }
}
```

- 如果钩子函数不返回 Promise 对象，但是有一个参数（transition），则钩子函数会等到 transition.next()、transition.abort()或 transition.redirect()之一被调用时才 resolve。代码示例如下：

```
route: {
  activate: function (transition) {
    // 1 秒后 resolve
    setTimeout(transition.next, 1000)
  }
}
```

- 在验证类钩子函数如 canActivate、canDeactivate 以及全局 beforeEach 钩子函数中，如果返回值是一个布尔值（Boolean），也会使得钩子函数被同步 resolve。

2. 在钩子函数中返回 Promise 对象

- 当在钩子函数中返回一个 Promise 对象时，系统会在该 Promise 被 resolve 之后自动调用 transition.next。
- 如果 Promise 在验证阶段被 reject，系统会调用 transition.abort。

- 如果 Promise 在激活阶段被 reject，系统会调用 transition.next。
- 对于验证类钩子函数（canActivate 和 canDeactivate），如果 Promise 被 resolve 之后的值是假值（falsy value），系统会中断此次切换。
- 如果一个被 reject 的 Promise 抛出了未捕获的异常，这个异常会继续向上抛出，除非在创建路由器时启用了参数 suppressTransitionError。

3. 钩子函数合并

和组件本身的生命周期钩子函数一样，以下路由生命周期钩子函数：

- data
- activate
- deactivate

也会在合并选项时（扩展类或使用 mixins）被合并。举例来说，如果组件本身定义了一个路由 data 钩子函数，而这个组件所调用的 mixin 也定义了一个路由 data 钩子函数，则这两个钩子函数都会被调用，并且各自返回的数据将会被最终合并到一起。

需要注意的是，验证类钩子函数如 canActivate、canDeactivate 和 canReuse 在合并选项时会直接被新值覆盖。

4. 钩子函数介绍

（1）canReuse

该钩子函数用来判断组件是否可以重用。若可以重用，则不会创建新的组件实例，只会调用 data 钩子函数来更新数据，而不会调用其他钩子函数；若不可重用，则会创建新的组件实例，此时各个钩子函数都会被调用。

canReuse 可以是一个布尔值或者同步返回布尔值的函数，默认值为 true。当其为函数时，可接受一个参数：

- transition（可选，对象）

在 canReuse 钩子函数中只能访问 transition.to 和 transition.from，详见“transition 对象”部分。

必须返回布尔类型，其他等效假值会按 false 对待。

注意：实际上判断组件是否可重用不仅仅依赖该钩子函数的值，当根路径不一样时，不论 canReuse 钩子函数是否可重用，组件都会被重新实例化，各个阶段的钩子函数都会被调用。只有当前路径和目的路径有共同的根路径时，canReuse 钩子函数才会起作用。

（2）canActivate

在验证阶段，当一个组件将要被切入时被调用。该函数可以接受一个参数：

- transition（可选，对象）

调用 transition.next()可以 resolve 该钩子函数，调用 transition.abort()可以取消此次切换。详见“transition 对象”部分。

可以返回 Promise 对象，以下是等价的 Promise 和 transition 操作：

```
resolve(true) -> transition.next()
resolve(false) -> transition.abort()
reject(reason) -> transition.abort(reason)
```

或者返回布尔值，以下是等价的布尔值和 transition 操作：

```
true -> transition.next()
false -> transition.abort()
```

该钩子函数的调用顺序是从上至下。子级组件视图的 canActivate 钩子函数仅在父级组件的 canActivate 被断定（resolved）之后调用。

（3）activate

在激活阶段，当组件被创建而且将要切换进入时被调用。该函数可以接受一个参数：

- transition（可选，对象）

调用 transition.next()可以 resolve 该钩子函数，调用 transition.abort()不可以取消此次切换，因为在执行到该钩子函数时验证已经合法了。详见“transition 对象”部分。

可以返回 Promise 对象，以下是等价的 Promise 和 transition 操作：

```
resolve -> transition.next()
reject(reason) -> transition.abort(reason)
```

该钩子函数用来控制视图切换和数据获取的时机。在该钩子函数被 resolve 之后，会同时进行视图的切换和 data 钩子函数的调用。

该钩子函数的调用顺序是从上至下。子级组件视图的 activate 钩子函数仅在父级组件的 activate 被断定（resolved）之后调用。

（4）data

在激活阶段，activate 钩子函数被 resolve 时调用，用于加载和设置当前组件的数据。该函数可以接受一个参数：

- transition（可选，对象）

调用 transition.next(data)会为组件的 data 相应属性赋值。例如使用{ a: 1, b: 2 }，路由会调用 component.$set('a', 1)以及 component.$set('b', 2)。

可以返回 Promise 对象，以下是等价的 Promise 和 transition 操作：

```
resolve(data) -> transition.next(data)
reject(reason) -> transition.abort(reason)
```

data 钩子函数会在 activate 被断定（resolved），以及界面切换之前被调用。切换进来的组件会得到一个名为$loadingRouteData 的元属性，其初始值为 true，在 data 钩子函数被断定后会被赋值为 false。这个属性可用来对切换进来的组件展示加载效果。

data 和 activate 钩子函数的不同之处在于：

data 在每次路由变动时都会被调用，即使当前组件可以被重用，但是 activate 仅在组件是新创建时才会被调用。

假设有一个组件对应于路由/message/:id，当前用户所处的路径是/message/1。当用户浏览/message/2 时，当前组件可以被重用，所以 activate 不会被调用。但是我们需要根据新的 id 参数去获取和更新数据，所以在大部分情况下，在 data 中获取数据比在 activate 中更加合理。

activate 的作用是控制切换到新组件的时机。data 钩子函数会在 activate 被断定（resolved）以及界面切换之前被调用，所以数据的获取和新组件的切入动画是并行进行的，而且在 data 被断定（resolved）之前，组件会处于“加载”状态。

从用户体验的角度来看一下两者的区别：

如果等到获取到数据之后再显示新组件，用户会感觉在切换前界面被卡住了；相反（指不用等到获取数据后再显示组件），应立刻响应用户的操作，切换视图，展示新组件的“加载”状态。如果我们在 CSS 中定义好相应的效果，这正好可以用来掩饰数据加载的时间。

这么说的话，如果想等到数据获取之后再切换视图，则可以在组件定义路由选项时，添加 waitForData: true 参数。代码示例如下：

```
// 调用 transition.next
route: {
  data: function (transition) {
    setTimeout(function () {
      transition.next({
        message: 'data fetched!'
      })
    }, 1000)
  }
}

<!-- 在模板中使用 $loadingRouteData -->
<div class="view">
  <div v-if="$loadingRouteData">Loading ...</div>
    <div v-if="!$loadingRouteData">
    <user-profile user="{{user}}"></user-profile>
    <user-post v-repeat="post in posts"></user-post>
  </div>
</div>
```

（5）canDeactivate

在验证阶段，当一个组件将要被切出时被调用，用来验证该组件是否可以被卸载。该函数可以接受一个参数：

- transition（可选，对象）

调用 transition.next()可以 resolve 该钩子函数，调用 transition.abort()可以取消此次切换。详见“transition 对象”部分。

可以返回 Promise 对象，以下是等价的 Promise 和 transition 操作：

```
resolve(true) -> transition.next()
resolve(false) -> transition.abort()
reject(reason) -> transition.abort(reason)
```

或者返回布尔值，以下是等价的布尔值和 transition 操作：

```
true -> transition.next()
false -> transition.abort()
```

该钩子函数的调用顺序是从下至上。组件的 canDeactivate 钩子函数仅在子级组件的 canDeactivate 被断定（resolved）之后调用。

（6）deactivate

在激活阶段，当一个组件将要被禁用和移除时被调用。该函数可以接受一个参数：

- transition（可选，对象）

调用 transition.next()可以 resolve 该钩子函数，调用 transition.abort()不可以取消此次切换，因为在执行到该钩子函数时验证已经合法了。详见“transition 对象”部分。

可以返回 Promise 对象，以下是等价的 Promise 和 transition 操作：

```
resolve -> transition.next()
reject(reason) -> transition.abort(reason)
```

在该钩子函数被 resolve 之后，新组件的 activate 钩子函数会被调用。

该钩子函数的调用顺序是从下至上。父级组件的 deactivate 钩子函数仅在子级组件的 deactivate 被断定（resolved）之后调用。

14.6 路由匹配

vue-router 做路径匹配时支持动态片段、全匹配片段以及查询参数（片段指的是 URL 中的一部分）。对于解析过的路由，这些信息都可以通过路由上下文对象（从现在起，我们会称其为路由对象）访问。在使用了 vue-router 的应用中，路由对象会被注入每个组件中，赋值为 this.$route，并且当路由切换时，路由对象会被更新。

14.6.1 动态片段

动态片段使用以冒号开头的路径片段定义，例如在 user/:username 中，:username 就是动态片段，它会匹配注入/user/foo 或者/user/bar 之类的路径。当路径匹配一条含有动态片段的路由规则时，动态片段的信息可以从$route.params 中获得。代码示例如下：

```
router.map({
  '/user/:username': {
    component: {
```

```
      template: '<p>用户名是{{$route.params.username}}</p>'
    }
  }
})
```

一条路径中可以包含多个动态片段，每个片段都会被解析成$route.params 的一个键值对。

14.6.2　全匹配片段

动态片段只能匹配路径中的一部分，而全匹配片段则基本类似于它的贪心版。例如，/foo/*bar 会匹配任何以/foo/开头的路径。匹配的部分也会被解析成$route.params 的一个键值对。

因为“*”可以匹配任何路径，所以我们可以用它来匹配默认路由。代码示例如下：

```
router.map({
  '*': {
    component: {
      template: '<p>default view</p>'
    }
  }
})
```

当所有路由都不匹配时，会默认匹配“*”对应的路由。

14.6.3　具名路径

在有些情况下，给一条路径加上一个名字能够让我们更方便地进行路径跳转。我们可以按照下面的示例给一条路径加上名字：

```
router.map({
  '/user/:userId': {
    name: 'user', // 给这条路径加上一个名字
    component: { ... }
  }
})
```

可以用 v-link 链接到该路径，代码示例如下：

```
<a v-link="{ name: 'user', params: { userId: 123 }}">User</a>
```

同样的，也可以用 router.go() 切换到该路径，代码示例如下：

```
router.go({ name: 'user', params: { userId: 123 }})
```

以上两种情况，路由最终都会切换到 /user/123。

14.6.4 路由对象

路由对象$route 存储了当前路由的信息，包括以下属性：

- path —— 字符串，等于当前路由对象的路径，会被解析为绝对路径，如"/foo/bar"。
- params —— 对象，包含路由中的动态片段和全匹配片段的键值对，详情见 14.6.1 节和 14.6.2 节。
- query —— 对象，包含路由中查询参数的键值对。例如，对于/foo?user=1，会得到 $route.query.user == 1。
- router —— 管理当前路由器的 vue-router 实例。
- matched —— 数组，包含当前匹配的路径中所包含的所有片段所对应的配置参数对象。
- name —— 字符串，当前路径的名字（请参阅 14.6.3 节）。

除了内置属性，在用 router.map 配置路由映射规则时定义的字段也会复制到最终的路由对象上。代码示例如下：

```
router.map({
  '/user': {
    auth: true,
    component: {
      template: '<p>用户名是{{$route.params.username}}</p>'
    }
  }
})
```

在 /user 被匹配时，当前路由对象$route.auth 的值是 true。基于此，我们可以在全局钩子函数中进行身份验证。代码示例如下：

```
router.beforeEach(function (transition) {
  if (transition.to.auth) {
    // 对用户身份进行验证...
  }
})
```

14.7　transition 对象

每个钩子函数都会接受一个 transition 对象作为参数。transition 对象并不是动画对象，它提供了很多方法来控制路由切换的时机。该对象包含以下属性：

- to（路由对象）—— 表示将要切换到的路由对象。关于路由对象请参阅 14.6.4 节。
- from（路由对象）—— 表示当前路径的路由对象。
- next（函数）—— 调用此函数处理切换过程的下一步。
- abort（函数）—— 调用此函数来终止或者拒绝此次切换。
- redirect（函数）—— 取消当前切换并重定向到另一个路由。

14.8　嵌套路由

嵌套路由和嵌套组件之间的匹配是一个很常见的需求，使用 vue-router 可以很简单地实现这个需求。

假设有一个应用，代码示例如下：

```
<div id="app">
  <router-view></router-view>
</div>
```

<router-view>是一个顶级的外链，它会渲染一个和顶级路由匹配的组件。代码示例如下：

```
router.map({
  '/foo': {
    // 路由匹配到/foo 时，会渲染一个 Foo 组件
    component: Foo
  }
})
```

同样的，在组件内部也可以包含自己的外链，嵌套的<router-view>。例如，我们在组件 Foo 的模板中添加了一个<router-view>，代码示例如下：

```
var Foo = Vue.extend({
  template:
    '<div class="foo">' +
```

```
    '<h2>This is Foo!</h2>' +
    '<router-view></router-view>' + // <- 嵌套的外链
  '</div>'
})
```

为了能够在这个嵌套的外链中渲染相应的组件，我们需要更新路由配置。代码示例如下：

```
router.map({
  '/foo': {
    component: Foo,
    // 在/foo 下设置一个子路由
    subRoutes: {
      '/bar': {
        // 当匹配到/foo/bar 时，会在 Foo 的 <router-view>内渲染
        // 一个 Bar 组件
        component: Bar
      },
      '/baz': {
        // Baz 也是一样的，不同之处是匹配的路由会是/foo/baz
        component: Baz
      }
    }
  }
})
```

使用以上配置，当访问/foo 时，Foo 的外链中不会渲染任何东西，因为在配置中没有任何子路由匹配这个地址。或许我们想渲染一些内容，此时可以设置一个子路由匹配“/”。代码示例如下：

```
router.map({
  '/foo': {
    component: Foo,
    // 在/foo 下设置一个子路由
    subRoutes: {
      '/': {
        // 当匹配到/foo 时，会在 Foo 的<router-view>内渲染
        component: Default
      },
      '/bar': {
        // 当匹配到/foo/bar 时，会在 Foo 的 <router-view>内渲染
        // 一个 Bar 组件
```

```
        component: Bar
      },
      '/baz': {
        // Baz 也是一样的，不同之处是匹配的路由会是/foo/baz
        component: Baz
      }
    }
  }
})
```

14.9　动态加载路由组件

当我们使用 Webpack 或者 Browserify 时，在基于异步组件编写的 Vue 项目中也可以较为容易地实现惰性加载组件。不再是之前所述的直接引用一个组件，现在需要像下面这样通过定义一个函数返回一个组件。代码示例如下：

```
router.map({
  '/async': {
    component: function (resolve) {
      // MyComponent 通过 RequireJS 从服务端加载
      resolve(MyComponent)
    }
  }
})
```

Webpack 已经集成了代码分割功能，我们可以使用 AMD 风格的 require 来标识代码分割点。代码示例如下：

```
require(['./MyComponent.vue'], function (MyComponent) {
  // 执行 MyComponent 组件加载完后的逻辑
})
```

和路由配合使用，代码示例如下：

```
router.map({
  '/async': {
    component: function (resolve) {
      require(['./MyComponent.vue'], resolve)
    }
  }
})
```

现在，只有当/async 需要被渲染时，MyComponent.vue 组件才会自动加载它的依赖组件，并且异步加载进来。

14.10　实战

在学习了 vue-router 应用的基本组成之后，现在我们运用所学到的知识来进行实战。首先我们会看一个直接在浏览器中引用相关文件的完整实例，接下来会基于 Webpack 构建工具做一个动态加载路由组件的高级实例。

14.10.1　浏览器直接引用

为了便于理解，我们会在一个 HTML 文件中完成实例分析。代码示例如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>vue router demo</title>
  <style>
    .v-link-active {
      color: green;
    }
  </style>
</head>
<body>
<div id="app"></div>
<script src="/vue/1.0.24/vue.min.js"></script>
<script src="/vue-router/0.7.13/vue-router.min.js"></script>
<!-- 根组件模板 -->
<script type="text/vue-tempalte" id="rootTemplate">
  <div class="wrapper">
    <div class="nav">
      <a v-link="{path: '/home'}">Home</a>
      <a v-link="{path: '/about'}">About</a>
    </div>
    <div class="router-view">
      <router-view></router-view>
  </div>
```

```
</div>
</script>
<script>
  // 路由组件定义，路由组件可以是通过 Vue.extend 创建的构造函数，或者是一个组件配置对象
  // 当为组件对象时，vue-router 内部会自动调用 Vue.extend 创建构造函数
  // 构造函数形式
  var Home = Vue.extend({
    template: 'This Is Home Page'
  })
  // 组件配置对象形式
  var About = {
    template: 'This Is About Page'
  }
  // 根组件，VueRouter 实例会用此根组件来启动应用
  var App = Vue.extend({
    template: document.querySelector('#rootTemplate').textContent
  })
  // 实例化 VueRouter
  var router = new VueRouter()
  // 路由映射配置
  router.map({
    '/home': {
      component: Home
    },
    '/about': {
      component: About
    }
  })
  /*
    以上路由映射等价于通过此种方式调用，实际上调用
    router.map 时在 vue-router 内部会对每个键值对
    调用 router.on 方法来完成路由规则映射
    router.on(
      '/home',
      {
        component: Home
      }
    })
    router.on(
```

```
      '/about',
      {
        component: About
      }
    )
  */
  // 默认访问重定向到/home
  router.redirect({
    '/': '/home'
  })
  router.start(App, '#app')
</script>
</body>
</html>
```

14.10.2　Webpack 模块化开发

当项目变得非常复杂时，很有必要采用模块化方式来开发。通常我们会用 Webpack 将各路由组件直接打包到一个文件中，实际上这种方式会导致不必要的代码被加载，因为有些模块用户可能不会访问到。

对于以上问题，我们可以结合 Webpack 的代码分割以及 Vue 组件的 resolve 来实现只有当用户访问某一个路由时，才去服务端获取相应的组件代码。这样做可以加快页面渲染速度，同时可以节省流量。下面我们来看看具体如何实现以上需求。

首先我们来看看目录结构：

```
example
  |- components
    |- about.vue
    |- home.vue
    |- not-found.vue
  |- app.vue
  |- index.html
  |- index.js
  |- router-config.js
  |- package.json
  |- webpack.config.js
```

各目录及文件含义如下：

- components 目录用来存放各个路由组件模块。
- app.vue 是应用的根组件，vue-router 用该组件来启动路由应用。
- index.html 是项目的入口 HTML 文件。
- index.js 是项目的入口 JS 文件。
- package.json 是 npm 包信息文件。
- router-config.js 包含路由配置规则。
- webpack.config.js 是 Webpack 构建工具的配置文件。

接下来我们按源码执行顺序来进行解析。

index.html 路由应用的入口页面，定义了根组件挂载的元素，以及引入 Webpack 打包后的 JavaScript 文件。代码示例如下：

```
<!-- example/index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>vue-loader advanced example</title>
</head>
<body>
  <!-- 用来挂载根元素 -->
  <div id="app"></div>
  <script src="/build/build.js"></script>
</body>
</html>
```

build.js 是源码打包后的入口文件，实际上源码入口是 index.js。

index.js 用来引入 Vue.js 和 vue-router，实例化并启动路由应用，是整个应用的 JS 入口。

```
// example/index.js
// 注意：依赖 Vue.js 0.12.10 及以后版本
// 加载 Vue.js 和 vue-router
var Vue = require('vue');
var VueRouter = require('vue-router');
```

```
var configRouter = require('./router-config');
// 这里必须调用 Vue.use 方法来安装 vue-router 插件，因为使用 CommonJS 规范
// Vue 对象不会暴露为 window 对象的属性，因此 vue-router 没法自动安装
Vue.use(VueRouter)
// 创建 router 实例
var router = new VueRouter({
});
// 用来配置路由映射规则
configRouter(router)
// 启动主组件，以此来启动整个路由应用
var App = require('./app.vue');
router.start(App, '#app');
```

router-config.js 用来配置路由映射规则、重定向、别名、全局钩子函数等，并利用 Webpack 的代码分割功能和 vue-router 的组件动态 resolve 的功能提供组件的动态加载。

注意源码中如 require（['./components/home.vue'], resolve）是 Webpack 提供的代码分割语法，采用 AMD 形式：

```
require(['./asyncModule'], function (module){
  module.doSomething()
  …
})
```

也就是说，Webpack 在编译时遇到以上 AMD 语法，并不会直接加载 asyncModule 源码，而是在浏览器中执行到此处时，才会去服务端动态加载，加载完后会执行回调方法，回调接受的参数是请求的 module。

在 vue-router 中，假如组件定义时 component 字段是一个方法，而不是对象或 Vue 组件构造函数，那么此方法的参数是一个 resolve 方法。结合 Webpack 的代码分割功能，将 resolve 作为 require 的回调，那么当用户访问一个本地没有的路由组件的路由时，Webpack 首先会去服务端请求该路由组件模块，请求完之后会执行 resolve 回调，在 resolve 回调中，vue-router 会将该组件渲染出来。

```
// example/router-config.js
module.exports = function configRouter (router) {
  // 定义路由映射规则
  router.map({
    // 常规组件（一级组件）
    '/home': {
```

```
    component: function (resolve) {
      require(['./components/home.vue'], resolve)
    }
  },
  // 常规组件（一级组件）
  '/about': {
    component: function (resolve) {
      require(['./components/about.vue'], resolve)
    }
  },
  // 未匹配时的路由组件
  '*': {
    component: function (resolve) {
      require(['./components/not-found.vue'], resolve)
    }
  }
 })
 // 路由别名配置
 router.alias({
  '/home/alias': '/home'
 })
 // 路由重定向
 router.redirect({
  '/': '/home'
 })
 // 全局前置钩子函数，在路由开始切换前执行，优先于路由组件钩子函数
 router.beforeEach(function (transition) {
  // transition.to 表示将要跳转到的路由对象
  // transition.from 表示当前路由对象
  if (transition.to.path === '/forbidden') {
    router.app.authenticating = true
    setTimeout(() => {
      router.app.authenticating = false
      // 终止路由切换，应用保持
      transition.abort()
    }, 3000)
  } else {
    transition.next()
  }
 })
}
```

在以上路由定义中，require 了根目录下的 app.vue 和 components 目录下的 home.vue、about.vue、not-found.vue 四个组件定义文件，每个文件都代表一个路由组件模块。下面我们看看路由组件的定义。

app.vue 定义了根组件，该组件定义了 v-link 导航部分以及根视图 router-view，一级组件都会渲染到该 router-view 中。

```
// example/app.vue
<style>
.v-link-active {
  color: green;
}
[v-cloak] {
  display: none;
}
</style>
<template>
  <div class="wrapper">
    <div class="nav">
      <a v-link="{path: '/home'}">Home</a>
      <a v-link="{path: '/about'}">About</a>
    </div>
    <div class="router-view">
      <router-view></router-view>
    </div>
  </div>
</template>
<script>
  module.exports = {}
</script>
```

home.vue、about.vue、not-found.vue 三个都是路由组件，是平行关系，这里主要以 home.vue 组件来进行分析。

home.vue 组件匹配 path 值为 /home。在组件的配置对象中，除了常规 Vue.js 组件的属性外，还可以传入路由配置对象 route 属性，用来控制路由切换过程中的各个阶段。route 是一个对象，它可以接受 5 个钩子函数属性分别用来控制切换时的 5 个阶段。

```
// example/components/home.vue
<template>
  This Is {{title}} Page
</template>
<script>
// 组件配置对象
module.exports = {
  // 路由配置对象
  route: {
    /**
     * 判断组件是否可重用
     * 只有当前组件和要切换到的组件的根路径相同时才会调用该钩子函数
     */
    canResuse: function () {
      return true;
    },
    /**
     * 从其他路由切换到该组件时，会调用此钩子函数判断
     * 可否切换到该组件，当该组件返回 true 或者被 resolve 时
     * 才会认为可切换到此组件，否则将会取消路由切换，同时
     * 停止执行之后的钩子函数
     */
    canActivate: function () {
      return true;
    },
    /**
     * 被 resolve 后视图开始切换的同时会调用
     * 注意：当组件可重用时，不会触发该钩子函数
     */
    activate: function () {
      var self = this;
      this.timer = setTimeout(function (){
        self.counter += 1;
      }, 3000);
    },
    /**
     * activate 被 resolve 后，调用该函数获取数据
     */
    data: function () {
```

```
      return {
        counter: 0
      };
    },
    /**
     * canDeactivate 被 resolve 后调用该函数，可以在
     * 函数中进行一些组件移除前的清理工作，如移除
     * 定时器等
     */
    deactivate: function () {
      //清理 activate 钩子函数中的定时器
      clearTimeout(this.timer);
    }
  },
  // 组件初始化时数据
  data: function () {
    return {
      title: 'Home',
      counter: 0
    }
  }
}
</script>
```

14.11　常见问题解析

1. saveScrollPosition 不生效

有时候我们希望在路由视图切换后，切换进来的视图保持在最后一次停留的位置。根据官方文档，我们需要在实例化 vue-router 时设置 saveScrollPosition 参数值为 true。但是当设置该参数值为 true 时，点击 v-link 元素导航切换到新视图，发现视图并没有停留在最后一次出现的位置。

对于这个问题，我们需要知道：

- saveScrollPosition 只在 HTML5 模式下生效，在实例化 vue-router 时，需要同时设置 history 值为 true。代码示例如下：

```
var router = new VueRouter({
  saveScrollPosition: true,
```

```
  // 设置为 HTML5 模式
  history: true
})
```

- 视图位置重置只发生在用户点击浏览器后退按钮时，因为在 vue-router 中通过监听 HTML5 history 的 popstate 事件来重置视图位置。因此在点击 v-link 元素切换视图时，视图位置不会发生重置。

通过实际测试发现，在 HTML5 模式下尽管 saveScrollPosition 参数值为 false0，但是对于 Chrome 和 Firefox 浏览器在点击前进、后退按钮时视图位置也会被重置为最后一次出现的位置（浏览器特性）。

2. 监听不到查询参数

有时候我们需要改变 hash 地址中的查询参数，比如从 #!/books/search?cat=1 到 #!/books/search?cat=2。这时我们希望根据新的 cat 参数重新向服务端获取数据。因为此时只有查询参数发生了变化，vue-router 认为组件是可重用的，所以只会执行 data 钩子函数。因此，我们需要在 data 钩子函数中执行相应的逻辑。代码示例如下：

```
var Search = Vue.extend({
  route: {
    data: function (transition) {
      console.log(this.$route.params.cat)
    }
  }
})
```

3. 判断当前路径和目的路径

有时候我们需要根据当前路径和将要跳转到的路径来决定下一步的操作，这时可以在全局钩子函数 beforeEach 中进行判断。代码示例如下：

```
var router = new VueRouter()
router.beforeEach(function (transition) {
  // transition.from 为当前路由对象
  if (transition.from.path === '/noleaving') {
    console.log('can not leave /noleaving')
    // 取消此次路由切换，停留在当前页面
    transition.abort()
    return
  }
  // transition.to 为目的路由对象
```

```
  if (transition.to.path === '/forbidden') {
    console.log('can not go to /forbidden')
    // 取消此次路由切换，停留在当前页面
    transition.abort()
    return
  }
  transition.next()
})
```

关于 transition 对象请参阅 14.7 节。

4. 切换路由时修改页面标题

通常在切换路由改变视图后需要更新页面标题，这里我们介绍一下使用 afterEach 全局钩子函数来更新页面标题的方法。代码示例如下：

```
// 路由组件定义，路由组件可以是通过 Vue.extend 创建的构造函数，或者是一个组件配置对象
// 当为组件对象时，vue-router 内部会自动调用 Vue.extend 创建构造函数
// 构造函数形式
var Home = Vue.extend({
  template: 'This Is Home Page'
});
// 组件配置对象形式
var About = {
  template: 'This Is About Page'
};
// 根组件，VueRouter 实例会用此根组件来启动应用
var App = Vue.extend({
  template: document.querySelector('#rootTemplate').textContent
});
// 实例化 VueRouter
var router = new VueRouter()
// 路由映射配置
router.map({
  '/home': {
    component: Home,
    // 组件对应的页面标题
    docTitle: 'Home Page'
  },
  '/about': {
    component: About,
    docTitle: 'About Page'
```

```
  }
})
// 该钩子函数在目的路由 canActivate 被 resolve 之后执行
router.afterEach(function (transition) {
  // transition.to 为目的路由映射对象
  document.title = transition.to.docTitle
})
// 启动 vue-router app
router.start(App, '#app')
```

5. canReuse 配置无效

在 14.5.1 节中我们了解到，第一阶段为组件可重用阶段。在这个阶段中，vue-router 首先会判断两个路由有无相同的根组件，如果没有则认为不可复用，而不管 canReuse 的值是否为 true。如果有相同的根组件，才会进一步判断用户设置的 canReuse 的值。判断组件是否可复用流程如图 14-6 所示。

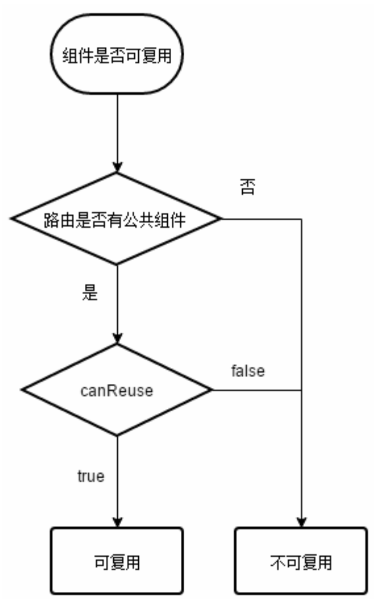


图14-6　判断组件是否可复用流程图

第 15 章 vue-cli

使用 Vue.js 开发大型应用时，我们需要考虑代码目录结构、项目构建和部署、热加载、代码单元测试等事情。如果每个项目都要手动完成这些工作，那无疑效率是低下的，所以通常我们会使用一些脚手架工具来帮助完成这些事情。在 Vue.js 生态中我们可以使用 vue-cli 脚手架工具来快速构建项目。

15.1 安装

vue-cli 是用 node 编写的命令行工具，我们需要进行全局安装。打开命令行终端，输入如下命令：

```
$ npm install -g vue-cli
```

注：*请确保 node 版本为 4.x、5.x 及以上。*

安装完成后执行如下命令：

```
$ vue -V
```

如果能显示 vue-cli 的版本号，则表示安装成功，如图 15-1 所示。

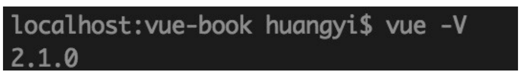


图15-1 查看vue-cli版本号

15.2 基本使用

我们可以使用 vue-cli 来快速生成一个基于 Webpack 构建的项目。打开命令行终端，输入如下命令：

```
$ vue init webpack my-project
```

执行命令后，会有一些命令行交互，我们可以初始化一些项目信息，如图 15-2 所示。

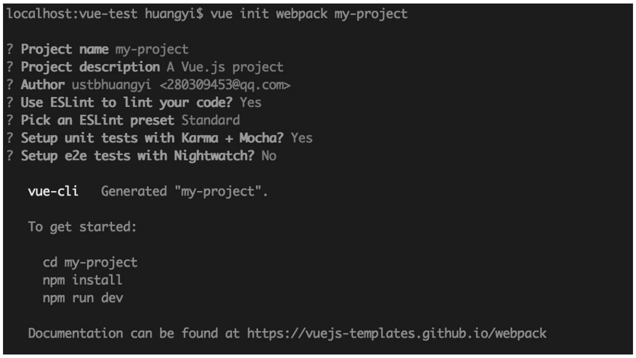


图15-2　vue-cli init命令初始化项目

项目初始化完成后，会在当前目录下生成 my-project 目录。进入 my-project 目录，安装项目的依赖，执行如下命令：

```
$ npm install
```

依赖安装完成后，我们来看一下项目的目录结构，如下所示：

```
|-my-project
  |- build              （构建脚本目录）
  |- config             （构建配置目录）
  |- node_modules       （依赖的 node 工具包目录）
  |-src                 （源码目录）
     |- assets          （资源目录）
     |- components      （组件目录）
     |- App.vue         （页面级 Vue 组件）
     |- main.js         （页面入口 JS 文件）
|- static               （静态文件目录）
|- test                 （测试文件目录）
|- indext.html          （入口页面）
|- .eslintrc.js         （ES 语法检查配置）
|- package.json         （项目描述文件）
|- …
```

我们启动项目，执行如下命令：

```
$ npm run dev
```

打开浏览器，输入 http://localhost:8080，生成的页面如图 15-3 所示。

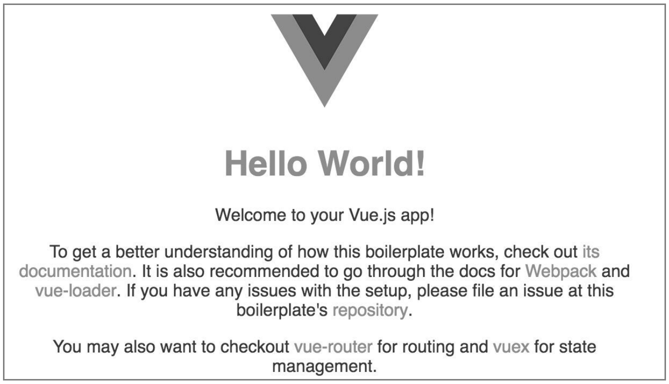


图15-3　vue-cli脚手架生成的初始页面

接下来我们打开 src/components/Hello.vue，修改一行代码，如下所示：

```
export default {
  data () {
    return {
      // note: changing this line won't causes changes
      // with hot-reload because the reloaded component
      // preserves its current state and we are modifying
      // its initial state.
      msg: 'Hello DDFE!'
    }
  }
}
```

然后刷新浏览器，可以看到内容发生了变化，如图 15-4 所示。

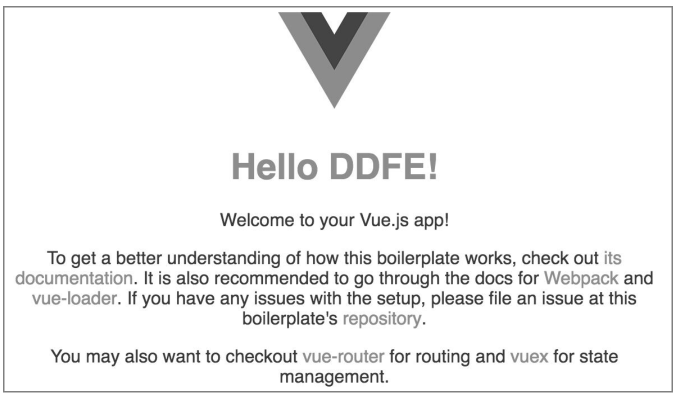


图15-4　vue-cli修改代码后的页面

我们可以根据项目需求做任何修改，可以在脚手架的基础上构建任何复杂的应用。

15.3　命令

vue-cli 安装后的全局命令是 vue，它支持以下两个子命令：

```
$ vue init <template-name><project-name>
$ vue list
```

15.3.1　init

init 命令用来基于指定模板生成项目结构。其中 template-name 为模板名，project-name 为要生成的目录名。关于模板请参阅 15.4 节。

15.3.2　list

init 命令用于列出所有可用的模板，通过查询 https://api.github.com/users/vuejs-templates/repos 这个 API 接口可以得到所有列表。源码实现如下：

```
<!--源码目录 bin/vue-list-->
request({
  url: 'https://api.github.com/users/vuejs-templates/repos',
  headers: {
    'User-Agent': 'vue-cli'
  }
}, function (err, res, body) {
  if (err) logger.fatal(err)
  console.log('  Available official templates:')
  console.log()
  JSON.parse(body).forEach(function (repo) {
    console.log(
      '  ' + chalk.yellow('★') +
      '  ' + chalk.blue(repo.name) +
      ' - ' + repo.description)
  })
})
```

15.4 模板

vue-cli 是一个项目脚手架工具，它支持通过模板来生成项目结构。在 15.3 节中，我们了解到在执行 init 命令时可以指定模板的名字。在默认情况下，vue-cli 会根据所传入的模板名字去 github 中查找模板。

vue-cli 的模板分为官方模板、自定义模板和本地模板。

注：*所有模板默认对应 2.0 版本，要安装 1.0 版本的模板，可以用 vue-cli init webpack-simple #1.0 my-project 这样的语法*。

15.4.1 官方模板

在使用之前，可以先用 vue list 命令查询都有哪些模板可供使用，之后通过 vue init 命令来生成相应模板的项目结构。模板分为基础和高级两个版本，其中基础版本用于快速构建原型；高级版本用于正式开发。所有模板都支持*.vue 组件。

目前官方提供了以下模板：

- Browserify —— 拥有高级功能的 Browserify + vueify 用于正式开发。
- browserify-simple —— 拥有基础功能的 Browserify + vueify 用于快速开发。
- browserify-simple-2.0 —— 拥有基础功能的 Browserify + vueify 用于 Vue.js 2.0 快速开发。
- simple —— 单个 HTML，用于开发最简单的 Vue.js 应用。
- simple-2.0 —— 单个 HTML，用于开发最简单的 Vue.js 2.0 应用。
- webpack —— 拥有高级功能的 Webpack + vue-loader 用于正式开发。
- webpack-simple —— 拥有基础功能的 Webpack + vue-loader 用于快速开发。
- webpack-simple-2.0 ——拥有基础功能的 Webpack + vue-loader 用于 Vue.js 2.0 快速开发。

15.4.2 自定义模板

当官方模板不能满足需求时，我们可以 fork 官方模板按照自己的需求修改后，通过 vue-cli 命令生成基于自己模板的项目结构：

```
$ vue init username/repo my-project
```

15.4.3　本地模板

除了从 github 下载模板外，我们还可以从本地加载模板：

```
$ vue init ~/fs/path/to-custom-template my-project
```

15.5　不错的工具包

vue-cli 内部使用了很多第三方 npm 包来帮助自己实现一些基础功能。接下来我们介绍其中一些不错的工具包。

15.5.1　commander

commander 是一个命令行接口的解决方案，它提供了一些接口方便我们对命令行的命令做解析。

仓库地址：https://github.com/tj/commander.js

15.5.2　download-git-repo

download-git-repo 用来将相应的 git 库（GitHub、GitLab、Bitbucket）下载到指定的本地文件夹。

仓库地址：https://github.com/flipxfx/download-git-repo

15.5.3　inquirer

inquirer 是一个常见的交互式命令行用户页面的集合，它可以简化以下流程：

- 提供错误反馈。
- 询问问题。
- 解析输入。
- 验证结果。

仓库地址：https://github.com/SBoudrias/Inquirer.js

15.5.4　ora

ora 可以让我们在终端展示加载效果，如图 15-5 所示。

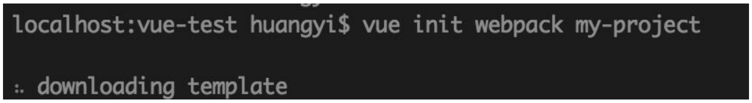


图15-5　ora加载效果

仓库地址：https://github.com/sindresorhus/ora

第 16 章

测试开发与调试

任何实际项目的开发都不仅仅是完成编码，规范的开发流程和严谨的测试都是不可或缺的。合理使用各种工具来进行测试开发与调试，能够极大地提升编写代码的效率，使开发过程事半功倍、对于提高代码质量、稳定线上服务至关重要。

Vue.js 除了是一个前端类库之外，还开发了许多配合使用的工具。比如 Chrome 下的调试工具、编辑器下的高亮工具等。正是这样一个完整的生态环境，使得用 Vue.js 开发变得更加简便。本章将为大家介绍几个常用的配合 Vue.js 使用的工具。

16.1 测试工具

16.1.1 ESLint

在日常的团队开发中，为了避免出现低级 bug 和统一代码风格，通常会在开发前约定一套编码规范。为了保证规范的执行，可以使用 Lint 工具和代码风格检测工具。

ESLint 就是一个 Lint 工具，它是由 JS 红宝书的作者 Nicholas C. Zakas 创立的一个开源项目，旨在为大家提供一个可扩展、每条规则独立、不内置编码风格的语法检查工具。ESLint 有别于 JSLint 的地方就是它被设计成完全可配置的，每一条规则都是一个插件，用户完全可以根据自己的需求来选择使用哪些规则。比如报错就可以设计为“警告”和“错误”两个等级，或者禁用。

下面介绍一下 ESLint 的配置。

在项目中配置 ESLint 有两种基本方法：

- 用 JavaScript 注解的方式将配置信息直接加到文件里。

- 使用 JavaScript、JSON 或 YAML 文件为整个目录定义配置信息。文件格式可以是 .eslintrc.*或 package.json。ESLint 会自动查找并读取配置文件。

需要配置的有以下几块信息，所有这些配置都将细粒度地决定 ESLint 如何检测代码。这里以 JSON 格式为例，展示一下基本的配置规则。

1. Enviroments

脚本将要运行的环境，每个环境都有自己预定义好的全局变量集合。通过 env 关键字配置 Environments 选项，下面的配置表示脚本将运行在浏览器（browser）和 node 环境。还可以配置 Commonjs、jQuery 等很多选项。

```
<!-- package.json -->
{
  "env":{
    "browser": true,
    "node": true
  }
}
```

2. Globals

在脚本运行期间需要额外加入的全局变量。当变量在当前文件中未定义却被访问时，会触发未定义规则警告。因此，如果设置了一些全局变量，则需要在 ESLint 的配置文件中进行配置。

```
<!-- package.json -->
{
  "globals": {
    "varA": true,
    "varB": false
  }
}
```

上述配置表明 varA、varB 都是全局变量，其中 varB 的值不可写（只读）。

3. Rules

ESLint 提供了大量的规则，用户通过配置规则是否生效来定义自己的项目需要使用哪些规则。

```
<!-- package.json -->
{
```

```
  "rules": {
    "eqeqe": "off",
    "curly": "error"
  }
}
```

16.1.2　工具包

1. eslint-loader

配合 Webpack 使用的 ESLint loader。

安装：

```
$ npm install eslint-loader
```

使用方法：

```
<!-- webpack.config.js-->
module.exports = {
  //…
  module:{
    loaders:[
      {test: /\.js$/, loader: "eslint-loader",exclude:/node_modules/}
    ]
  }
//…
}
```

当使用编译类的 loader（bable-loader）时要确保处理顺序正确，从下至上为处理顺序，所以语法检查要放在最下面。为保险起见，也可以使用 preLoaders 配置项，确保对源代码在没有被编译前进行检查。

2. eslint-friendly-formatter

这个组件可以友好地提示语法问题，并且支持点击错误提示直接打开 Sublime 或 iTerm2 文件。它可以被当作一个模块引用，还可以配合 gulp 或 grunt 使用。

3. eslint-config-standard

如果觉得自己配置很麻烦，也可以使用这个组件，它为大家提供了一些可共享的 JavaScript 标准格式配置。其用法也非常简单，在自己的工程中加入如下.eslintrc 文件。

```
<!-- .eslintrc 文件-->
{
  "extends": "standard"
}
```

4. eslint-plugin-html

一个支持从 HTML 等文件的<script>标签中读取配置的插件。通常其配置文件都是 JS 文件。

16.2 开发工具

16.2.1 Vue Syntax Highlight

开发 Vue.js 项目时还可能会遇到一个问题，就是.vue 后缀文件中的内容是不会被自动高亮显示的，感觉像在日记本里开发一样。不过没有关系，Vue.js 团队已经帮助我们开发了可以在 Sublime 中使用的插件——Vue Syntax Highlight。其安装方法如下：

```
sublime->preference->package-control->package-install->Vue Syntax Highlight
```

16.2.2 Snippets

Snippets 是一个帮助提高开发效率的小工具，通过它可以自动补全，我们只需输入一些简写的命令就可以自动生成代码。比如：

```
db + Tab -> debugger
cl +  Tab -> console.log
```

在 Sublime 中可以选择 Tools→New Snippet 来添加。在 github 上有许多别人配置好的文件可供参考。

```
<!--  Snippet 文件的基本格式 -->
<snippet>
  <content>
    <![CDATA[Hello, ${1:this} is a ${2:snippet}.]]>
  </content>
  <!-- <tabTrigger>hello</tabTrigger> -->
  <!-- <scope>source.python</scope> -->
</snippet>
```

<![CDATA[Hello, ${1:this} is a ${2:snippet}]]>标签中的是缩写补全后的样子，${1:this}是第

一个输入点，${2:snippet}是第二个输入点。补全后输入点之间可以通过 Tab 切换。

比如想快速生成一个包含 v-if 的 div 组件，可以进行如下配置：

```
<!--  snippet 文件配置示例 -->
<snippet>
  <content><![CDATA[
  <div v-if="${1}">${2}</div>
  ]]></content>
  <tabTrigger>v-if</tabTrigger>
  <scope>text.html</scope>
  <description>&lt;div v-if=""&gt;</description>
</snippet>
```

保存后，就可以通过输入 v-if + Tab 来获得<div v-if=' '></div>了。

16.2.3　其他编辑器/IDE

除了 Sublime Text，前端开发利器 WebStorm 以及微软新推出的风头正劲的编辑器 Visual Studio Code 也添加了对 Vue.js 的语法支持。二者同 Sublime Text 一样，也是通过安装插件的方式来识别以.vue 为后缀的文件的。

1. WebStorm

WebStorm 中用于支持 Vue.js 的插件名称就叫 Vue.js，安装方式是：Preferences→Plugins→Browse Repositories→Vue.js→Install，如图 16-1 所示。

安装完 Vue.js 插件后，重启 WebStorm，可以发现，以.vue 结尾的文件已经能够被识别，出现语法高亮显示，如图 16-2 所示。

从图 16-2 可以看到，虽然 WebStorm 已经支持.vue 后缀文件的语法高亮显示，但是其对文件中的 ES 6 语法并不识别，解决方式是在 Preferences→Editor→Language Injections 中添加一项，类型为 XML Attribute Injection，配置如图 16-3 所示。

然后在<script>标签中写入 type 属性，如<script type="es6"></script>即可。如此便可使 WebStorm 完成对.vue 后缀文件的全部识别，如图 16-4 所示。

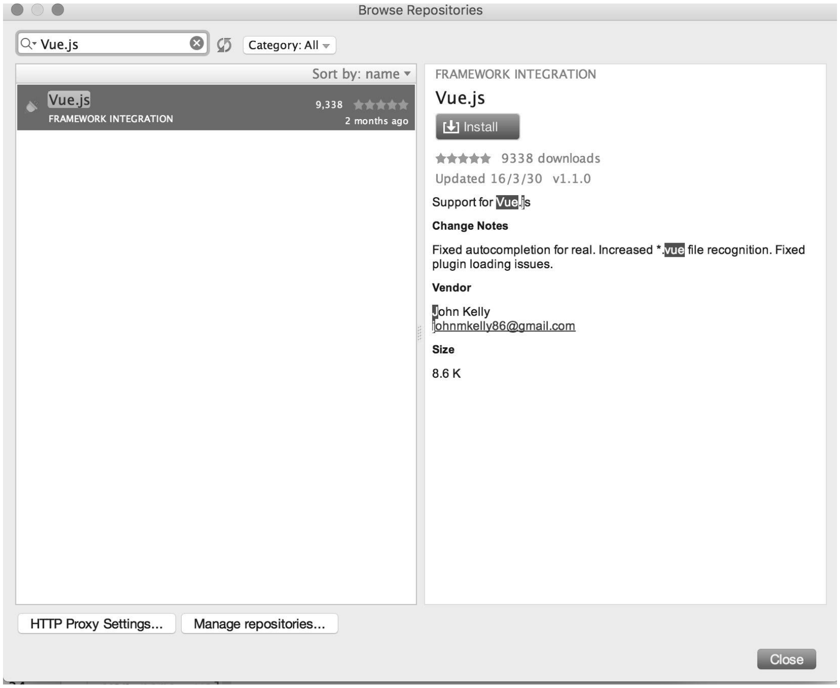


图16-1　WebStorm安装Vue.js插件界面

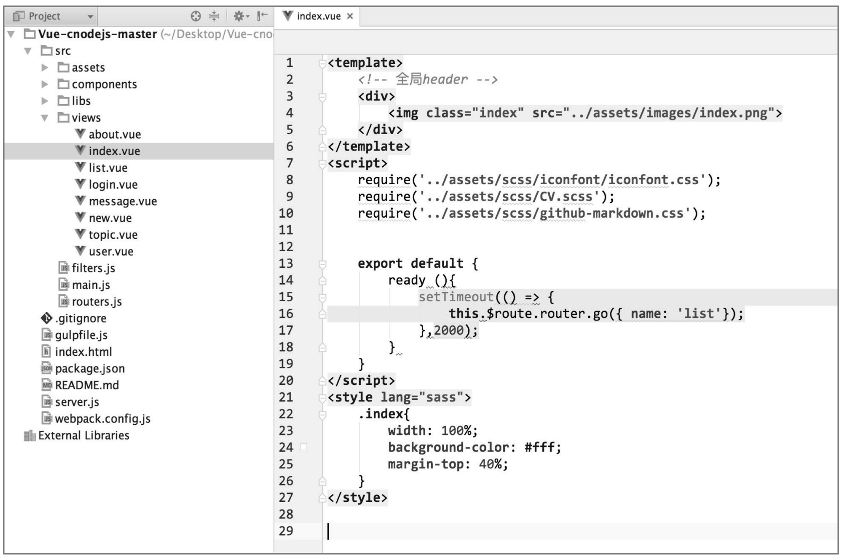


图16-2　WebStorm识别.vue后缀文件

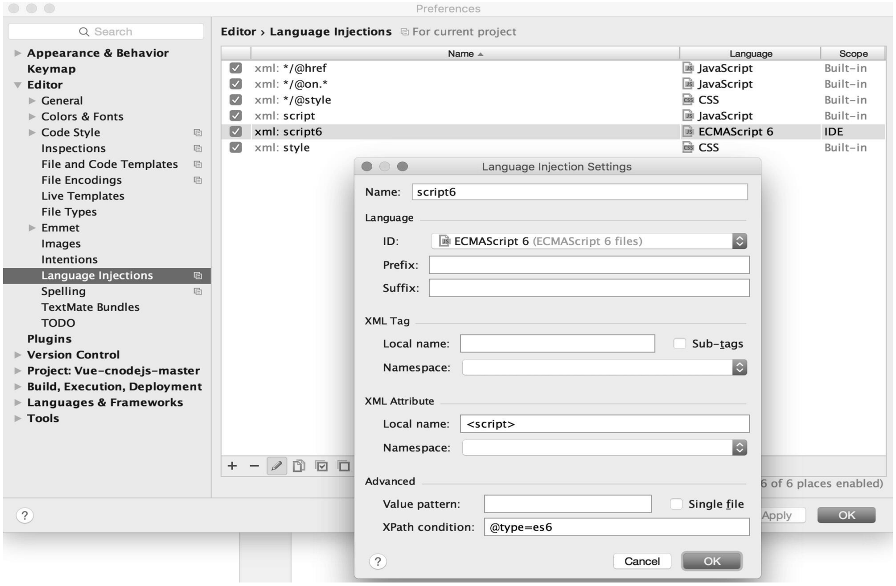


图16-3　WebStorm识别ES 6语法配置

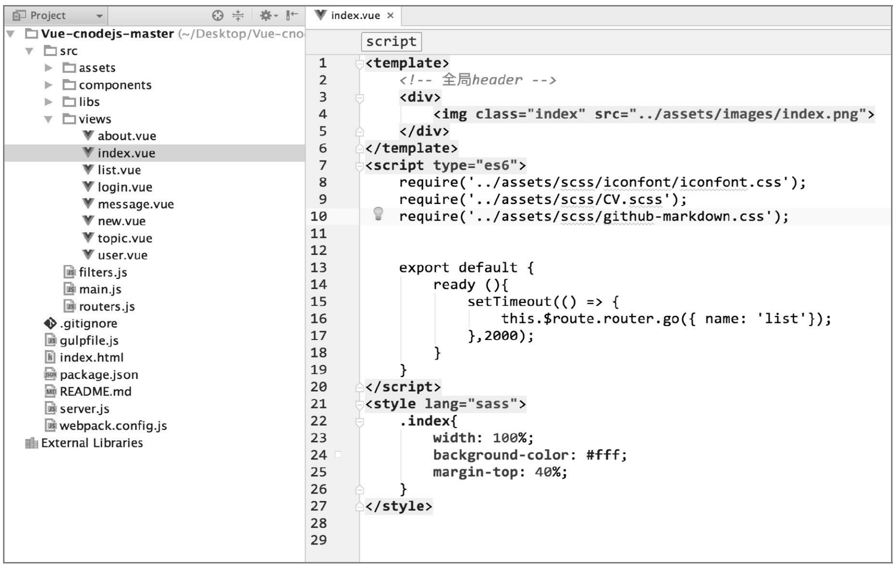


图16-4　WebStorm识别ES 6语法

除此之外，在 WebStorm 中对*.js 文件的 ES 6 语法支持设置方式是：Preferences→ Languages & Frameworks→JavaScript→JavaScript language version：ECMAScript 6，如图 16-5 所示。

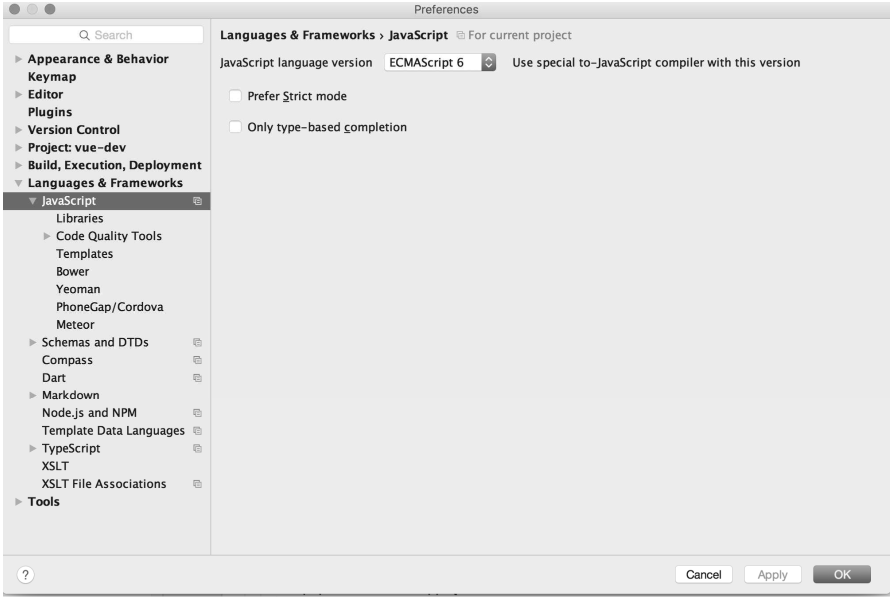


图16-5　WebStorm支持ES 6语法通用设置

2. Visual Studio Code

在 Visual Studio Code（以下简称 VSC）中为 Vue.js 提供语法高亮显示的插件名称为 vue，如图 16-6 所示。

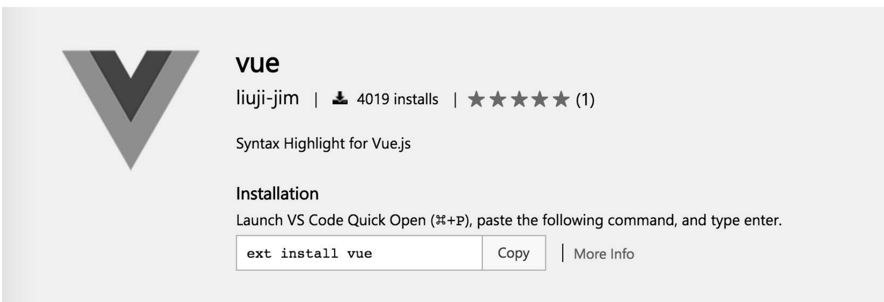


图16-6　VSC中的vue插件

vue 插件在 VSC 中的安装方式是，按“Cmd+P”快捷键呼出 VSC 命令行，然后键入“ext install vue”，按回车键，即可成功安装。

插件安装完成后，重启 VSC，插件即可生效，如图 16-7 所示。

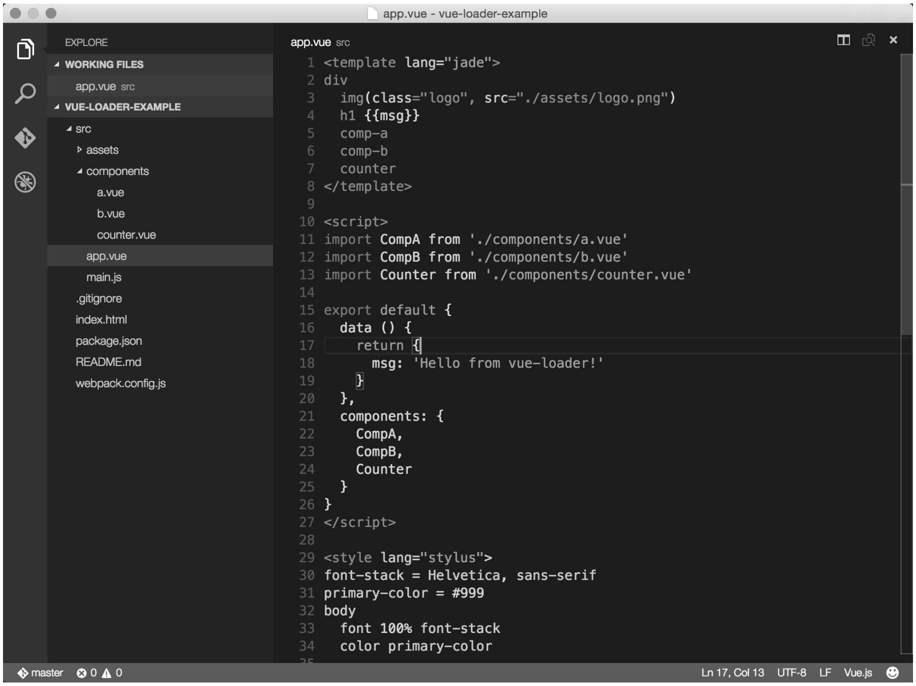


图16-7　VSC识别.vue后缀文件

16.3　调试工具

Vue.js 团队还为大家提供了 Chrome 下的调试工具，可以在 Chrome 的插件商店找到并安装，如图 16-8 所示。

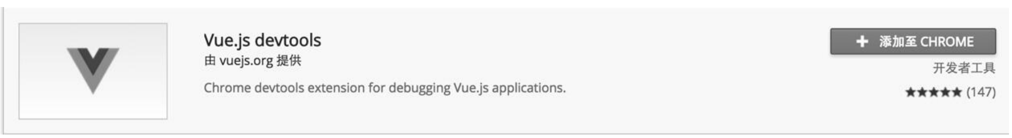


图16-8　Chrome的Vue.js调试插件——Vue.js devtools

安装成功后，在 Chrome 调试状态下的工具栏中就会出现 Vue Devtools 选项，选中后界面如图 16-9 所示。

该工具可以展示出各个组件的层级结构、组件当前的状态、组件的 prop 值。还可以通过点击 Inspect Dom 一键返回到 DOM 结构的查看页面。

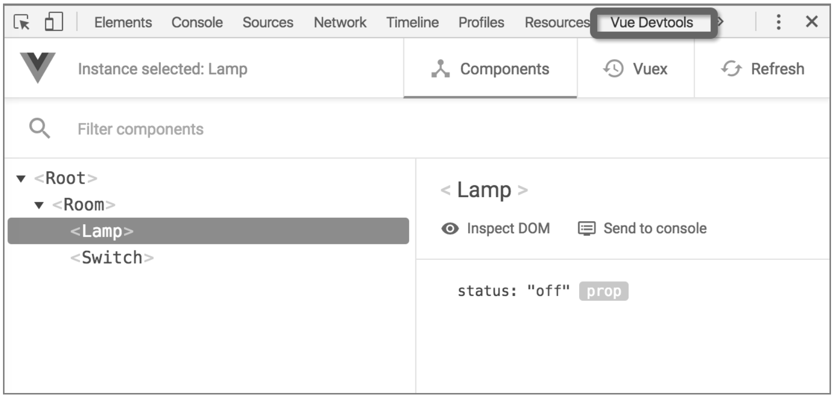


图16-9 Vue Devtools界面

如果无法翻墙使用 Chrome 的软件商店，也可以在 Vue.js 的 github 上找到源码自己编译安装，地址为 https://github.com/vuejs/vue-devtools。

第 17 章

Scrat+Vue.js 的化学反应

Vue.js 是一个小而美的库，非常适合开发一些 webapp 项目。我们团队 webapp 的技术栈是用 Scrat 进行模块化和构建的，自然会考虑到用 Scrat +Vue.js 做一套完整的移动端开发解决方案，使前端工程化+Vue.js 完美结合。既然提到前端工程化，那就让我们先来认识一下它。

17.1 浅谈前端工程化

现如今，前端可谓是包罗万象，产品形态五花八门，有小而美的前端基础库、酷炫的运营活动页、好玩的 h5 小游戏等。不过这些小项目并非是前端技术的主要应用场景。更具有商业价值的是复杂的 Web 应用，如新闻聚合网站、在线购物平台、社交网络平台、金融信贷应用、直播互动社区、打车出行软件等，它们功能完善、界面繁多、交互复杂，为用户提供了完整的用户体验。

这些复杂的 Web 应用，一般需要几十、上百人共同开发维护，其前端界面也颇具规模，工程量是非常庞大的，这就要求我们从软件工程的角度来思考前端开发，用工程化的手段来解决前端开发中遇到的各种问题，提升团队的开发效率。这就是所谓的前端工程化。

17.2 前端工程化怎么做

一些人可能认为，前端工程化无非就是库/框架选型+简单的构建化+JavaScript/CSS 模块化开发。其实这些只是工程化的一部分，当我们开发一个完整的 Web 应用时，将面临更多的工程

问题，如：如何多人协作开发、组件模块如何复用、如何调试部署、版本如何管理控制、性能如何优化等。因此，想做好前端工程化，需要做好以下几件事：

1. 开发规范

制定好开发、部署的目录规范、编码规范。好的目录规范能让项目结构清晰，便于维护和扩展；好的编码规范能让团队内同学的代码风格统一，便于代码审查。

2. 模块化

针对 JavaScript、CSS，以功能或业务为单元组织代码。JavaScript 模块化方案很多，如 AMD/CommonJS/UMD/ES 6 Module 等。CSS 模块化开发基本上都是在 less、sass、stylus 等预处理器的 import/mixin 特性支持下实现的。

3. 组件化

把页面拆分成多个组件（component），每个组件依赖的 CSS、JavaScript、模板、图片等资源放在一起开发和维护。组件是资源独立的，组件在系统内部可复用，组件和组件之间可以嵌套。现在流行的一些框架如 Polymer、React、Vue.js 等都提倡组件化开发方式。

4. 组件库

有了组件化，我们还希望把一些非常通用的组件或者 JavaScript 模块放到一个公共的地方供团队共享，方便新项目的复用，这就形成了组件库。常见的组件库有 bower、component 等。

5. 性能优化

通过工程化手段来解决性能优化问题。比如常见的请求合并、资源压缩、CDN，甚至一些前沿的优化手段如 bigpipe 和 bigrender，都是通过工程化手段来保证的，而对业务开发者是透明的。

6. 项目部署

项目部署一般包括静态资源缓存、CDN、增量发布等问题。合理的静态资源部署可以为前端性能带来较大的优化空间，而增量发布又为项目的版本控制、A/B Test 方案提供了保证。

7. 开发流程

完整的开发流程包括本地化开发调试、视觉走查确认、前后端联调、测试、上线等环节，通过一些工具对开发流程进行改善可以大幅度降低研发成本。

8. 工程工具

工程工具包括构建与优化工具、开发—调试—部署等流程工具、组件获取和提交工具等。这

些工具往往都是独立的系统，如果分开来用，成本太高了。因此能否串联这些功能，使得前端开发可以持续集成，工具的设计就变得至关重要了。接下来将要介绍一套前端集成解决方案——Scrat。它集合了这些工程工具，同时还集合了上述提到的 7 项前端工程化所需的技术要素。

最后，我们通过一张图再总结一下前端工程化所需的 8 大要素，它们之间并非孤立，而是存在一定的联系，如图 17-1 所示。

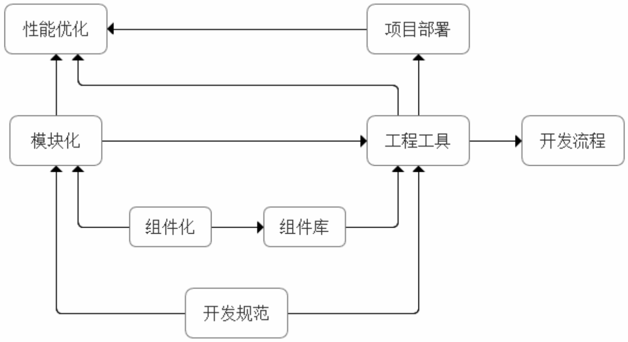


图17-1　前端工程化8大要素

17.3　Scrat 简介

Scrat 是 UC 团队在百度的 FIS 基础上二次开发的 webapp 模块化开发框架，它的功能非常强大，如图 17-2 所示。



图17-2　scrat-webapp模块化开发框架

Scrat 拥有前端工程化所需的所有能力，它最大的特色就是模块化开发和模块生态。Scrat 的理念是希望像搭积木一样开发和维护系统，通过组装模块得到一个完整的系统，这就是组件化开发。同时，每次研发新产品都不是从零开始，不同团队、不同项目能有可复用的模块沉淀下来，这就形成了组件库。

在 Scrat 中，静态资源分为模块化资源和非模块化资源两类，其中模块化资源还分为工程模块和生态模块两类，如图 17-3 所示。

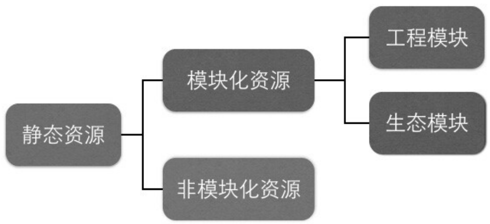


图17-3 静态资源划分

1. 模块化资源

模块化资源是具有独立性的模快所对应的静态资源。每个独立的模块都将自己所依赖的 JavaScript、CSS、模板、图片等资源放在一起维护，使得模块具备独立性，引用模块的 JavaScript 即可。其中，工程模块是当前工程所开发的模块，它们往往与业务耦合度较高；生态模块通常是一些通用性比较高的模块，基于 component 规范开发，部署到 github 上，可以通过 Scrat 命令安装到本地工程中。

2. 非模块化资源

在项目中并不是所有资源都应该被模块化，也有一些非模块化资源，一般为入口页面、模块化框架 JavaScript、第三方 JavaScript 库、页面启动 JavaScript 等。

其目录规范为：

```
project
  |- component_modules（生态模块）
  |- components        （工程模块）
  |- views             （非模块资源）
  |- …
```

17.4　Scrat+Vue.js 实现组件

Scrat 的模块化、组件化思想可以很好地与 Vue.js 配合，让我们来看看如何在 Scrat 项目中定义一个 Vue 组件。比如定义一个 header 组件，目录结构如下：

```
components
  |- header
  |- header.js
  |- header.styl
  |- header.tpl
    |- logo.png
```

然后在 header.js 中定义 Vue 组件，代码示例如下：

```
module.exports = Vue.extend({
  template: __inline('header.tpl'),
    data: function () {
      return {
        applink: 'http://d.xiaojukeji.com/c/71444'
      }
    }
});
```

这里我们借用了 FIS 的__inline 方法把 header.tpl 的内容内嵌到 header.js 中，经过 Scrat 编译后就变成了如下代码：

```
define('components/header/header.js', function(require, exports, module){
  module.exports = Vue.extend({
    template: "<div class=\"header\">\n  <div class=\"logo\"></div>\n  <div
class=\"download\">\n    <a :href=\"applink\">下载 APP</a>\n  </div>\n</div>",
    data: function () {
      return {
       applink: 'http://d.xiaojukeji.com/c/71444'
        }
      }
    });
});
```

在其他组件中想用这个组件可以这样写，代码示例如下：

```
new Vue({
  el: '#main',
```

```
  template: __inline('home.tpl'),
  components: {
    'v-header': require('header')    // 借助模块化使用组件并注册
  }
});
```

在 Scrat 中通过 require('header')可以自动加载组件依赖的同名的样式文件(CSS、stylus)等。最后，在 home.tpl 中就可以通过<v-header></v-header>来展现组件了。

上面简单介绍了如何在 Scrat 中定义一个 Vue 组件，接下来我们通过一个完整的案例看一下 Scrat 和 Vue.js 是如何配合使用的。

17.5　案例分析

我们需要做一个行程分享页面，页面中有司机信息、订单信息、地图等内容。订单状态是可变的，共有等待接驾、行程中、行程结束、行程取消、行程过期 5 种状态，随着订单状态的改变，页面的展示也随之发生变化，如图 17-4、图 17-5、图 17-6 所示。



图17-4　等待接驾、行程中、行程结束效果图



图17-5　行程取消效果图

我们拿到这样一个需求，如果不用 Vue.js，那么会在 JavaScript 中写大量的状态判断逻辑和 DOM 操作更新视图，稍不注意就会出错，这个过程是很痛苦的。然而，我们使用 Vue.js，实现这样的需求就很轻松了。下面我们详细介绍如何用 Scrat+Vue.js 开发这个项目。



图17-6　行程过期效果图

17.5.1　准备工作

1. 安装 Scrat

在命令行执行如下命令：

```
$ npm install -g scrat
```

安装后执行 scrat -v，查看工具版本，如图 17-7 所示。



图17-7　查看Scrat版本

如果安装失败，可以尝试淘宝镜像安装：

```
$ npm install -g cnpm --registry=https://registry.npm.taobao.org
$ cnpm install -g scrat
```

2. 初始化项目

Scrat 自带脚手架功能，通过执行 scrat init 命令生成一个 Scrat 脚手架项目，如图 17-8 所示。

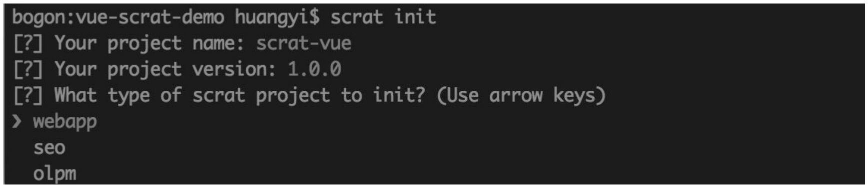


图17-8　Scrat脚手架

生成的目录结构如下：

```
vue-scrat-demo
  |-components          （模块化资源）
  |- server             （服务端代码）
  |- views              （非模块化资源）
  |- component.json     （模块化资源描述文件）
  |- fis-conf.js        （构建工具配置文件）
  |- package.json       （项目描述文件）
  |- …
```

3. 安装依赖组件库

Scrat 采用 component 作为生态模块，因此可以通过安装 component 组件，方便开发和团队共享。打开 component.json 文件，修改依赖关系，如下所示：

```
{
  "name": "scrat-vue",
  "version": "1.0.0",
  "dependencies": {
    "scrat-team/fastclick": "1.0.2"
  }
}
```

生态模块名称的结构是“用户名/仓库名@版本号”，这里的 fastclick 是一个 JavaScript 库，是由 github 用户 scrat-team 创建的，版本是 1.0.2。接下来我们在项目目录下执行 scrat install 命令，如图 17-9 所示。

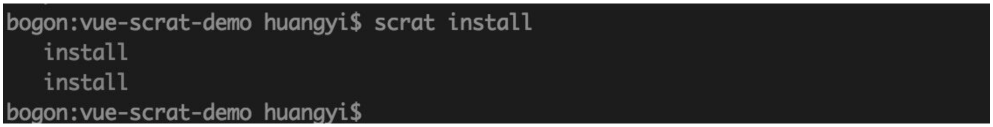


图17-9　Scrat安装依赖的生态模块

依赖的 fastclick 库会安装到当前项目中，目录结构如下：

```
vue-scrat-demo
  |- component_modules
  |- scrat-team-fastclick
    |- 1.0.2
      |- component.json
        |- fastclick.js
  |-components
  |- …
```

模块安装后，我们就可以在 JavaScript 代码中通过 require('fastclick')来引用这个模块了。

17.5.2　代码实现

1. 组件拆分

在做好准备工作后，开始正式编写代码前，我们可以依据功能需求，把页面拆分成一个个组件，如图 17-10、图 17-11、图 17-12 所示。



图17-10　组件拆分（header、order、map组件）



图17-11　组件拆分（cancel组件）



图17-12　组件拆分（expired组件）

我们把页面按功能拆分成多个组件后，很自然就可以梳理出组件目录结构（页面本身也可以看成一个组件——home 组件），如下所示：

```
|- vue-scrat-demo
  |- component_modules（模块生态资源）
  |- components
    |- cancel             （行程取消组件）
    |- common             （公共 js、stylus、image、font 资源）
    |- expired            （行程过期组件）
    |- header             （头部组件）
    |- home               （页面组件）
    |- map                （地图组件）
    |- order              （订单组件）
      |- order.js         （订单组件 js）
      |- order.styl       （订单组件 stylus）
      |- order.tpl        （订单组件模板）
  |- views
    |- lib                （scrat.js、vue.js、normalize.css 等非模块化资源）
    |- index.html         （入口页面）
  |- server               （服务端逻辑）
  |- fis.conf             （构建工具配置文件）
  |- …
```

2. 页面入口

页面入口为 index.html，引入了 lib.js（包括 scrat.js 和 Vue.js）。其中，scrat.js 是 Scrat 开发

体系的模块化框架，它与 Scrat 构建工具紧密结合，实现了 JavaScript/CSS 依赖管理、请求合并、按需加载、本地缓存等功能。同时，引入了 Vue.js，我们可以全局使用 Vue 对象。

页面是基于 JavaScript 渲染的，代码示例如下：

```
<script>
  require.config(__FRAMEWORK_CONFIG__);
  require.async(['home', 'fastclick'], function (home, fastclick) {
    fastclick(document.body);
    home.render(document.body);
  });
</script>
```

其中，__FRAMEWORK_CONFIG__这个全局宏会在项目构建时被替换成一个对象（包括模块描述、依赖关系、其他参数等）。Scrat 通过 require.config 接口知道了所有依赖关系，因此在 require.async 加载模块时，可以找到依赖的所有 JavaScript、CSS 模块并加载。当全部请求加载完成后触发 callback 函数。

页面渲染主要就是通过 home 组件的 render 方法实现的，接下来我们看一下 home 组件的实现。

3. home 组件的实现

home 组件通过 render 方法渲染页面主框架，代码示例如下：

```
/**
 * 渲染页面骨架
 * @param dom
 */
exports.render = function (selector) {
  new Vue({
    el: selector,
    replace: false,
    template: __inline('home.tpl'),
    data: {
      orderInfo: {
        orderStatus: 0,
        fromAddress: '北京南站',
        toAddress: '北京北站',
        getOnTime: 1464624221,
        getOffTime: 1464684221
```

```
      },
      driverInfo: {
        name: 'DDFE 司机',
            score: 5,
            car: 'DDFE 私家车',
            plate: '京 DDFE123',
            avatar: 'https://avatars3.githubusercontent.com/u/5359011 '
        }
    },
    events: {
      'order.change': function (orderInfo) {
      this.orderInfo = orderInfo;
        this.$broadcast('order.change', orderInfo);
    }
  },
  components: {
      'v-header': require('header'),
      'v-order': require('order'),
      'v-map': require('map'),
      'v-cancel': require('cancel'),
      'v-expired': require('expired')
    }
  });
};
```

我们通过 new Vue 创建了一个 Vue 实例，并把它挂载在 selector 对象上。由于传入的 selector 是 document.body，因此 replace 属性值要设置为 false。这里的 template 通过 FIS 特有的__inline 方法把 home.tpl 的内容内嵌到 home.js 中，从而实现了模板与 JavaScript 文件分离。在 data 属性中我们 mock 了一些数据，作为模板的数据源。在 events 属性中我们监听了 order.change 事件，该事件是由 order 组件派发的，home 组件接收后，广播给其他子组件。Components 中描述了 home 组件包含的所有子组件，可以看到，所有子组件都可以通过'v-xxx': require('xxx')方式加载并局部注册，这样我们就可以在模板中通过<v-xxx></v-xxx>来使用了。

接下来我们看一下 home.tpl 的实现，代码示例如下：

```
<v-header></v-header>
<div class="container">
  <v-order v-show="orderInfo.orderStatus!==4" :driver-info="driverInfo" :order- info=
"orderInfo"></v-order>
  <v-map v-show="orderInfo.orderStatus<3"></v-map>
```

```
  <v-cancel v-show="orderInfo.orderStatus===3"></v-cancel>
  <v-expired v-show="orderInfo.orderStatus===4"></v-expired>
</div>
```

在模板中，我们通过 v-show 指令结合订单状态字段 orderInfo.orderStatus 来判断隐藏或者显示不同的组件。这就是 Vue.js 带给我们的福利，不需要在 JavaScript 中写烦琐的判断逻辑和 DOM 操作，只需要在模板中通过一些指令和判断条件，就能实现根据不同的状态渲染不同的视图。这样不容易出错，极大地提升了研发效率。

可以看到，我们通过 v-bind 指令的简写方式:driver-info 和:order-info 把 data 中定义的 driverInfo 和 orderInfo 数据传递给 order 组件的 props。接下来我们看一下 order 组件的实现。

4. order 组件的实现

order 组件作为 home 组件的子组件，代码示例如下：

```
var api = require('common/js/api');
module.exports = Vue.extend({
  template: __inline('order.tpl'),
  props: ['driver-info', 'order-info'],
  created: function () {
    this.fetchOrderData();
  },
  methods: {
    fetchOrderData: function () {
      var me = this;
      api.getOrderInfo(function (orderInfo) {
        if (me.orderInfo.orderStatus !== orderInfo.orderStatus) {
          me.orderInfo = orderInfo;
              me.$dispatch('order.change', orderInfo);
        }
        setTimeout(function () {
            me.fetchOrderData();
          }, 5000);
      });
    }
  }
});
```

我们通过 Vue.extend 方法创建一个组件的构造器，并作为模块导出。这样在 home 组件中就可以通过 require('order')加载这个构造器，并把它当作 Vue 实例的 components 属性完成局部

注册。template 依旧通过 FIS 的__inline 方法加载 order.tpl 的内容。props 可以接收父组件传递的数据。created 为 Vue 实例生命周期中的一部分，在 Vue 实例化时调用。我们通过在 methods 中定义的 fetchOrderData 方法加载订单数据，这里的数据都是 mock 的，每 5s 请求一次。下面简单看一下服务端 mock 部分，代码示例如下：

```
var status = 0;
app.get('/orderInfo', function (req, res, next) {
  res.send({
    orderStatus: status,
    fromAddress: '北京南站',
    toAddress: '北京北站',
    getOnTime: 1464624221,
    getOffTime: 1464684221
  });
  if (status < 4) {
    status++;
  } else {
    status = 0;
}
});
```

服务端采用的是 node 的 Express 框架。当然，采用其他后端语言也是可以的，这本身就是一个前后端分离的项目，在这里只是 mock 数据。可以看到，服务端返回的数据每次 orderStatus 都会不一样，当 order 组件接收到不一样的 orderStatus 时，会通过$dispatch 方法派发这个 order.change 事件给它的父组件。home 组件会接收到这个事件，更新自己的 orderStatus，同时自身的视图也会根据 orderStatus 的改变自动刷新。home 组件还会广播这个事件，所有监听这个事件的子组件都可以收到并做相应的处理。

接下来还有 map、cancel、expired 组件等，它们的代码和 order 组件大同小异，限于篇幅这里就不一一介绍了。

17.5.3　编译和发布

项目代码开发完成后，需要编译和发布。在 Scrat 中，在项目目录下执行 scrat release 命令，即可对项目进行构建，并将构建结果发布到本地调试目录下，如图 17-13 所示。

```
bogon:vue-scrat-demo huangyi$ scrat release

 Ω                                          .... ..  124ms
bogon:vue-scrat-demo huangyi$
```

图17-13　Scrat构建项目

构建工具常见的功能就是区分构建目的，比如开发构建、测试构建、上线构建等。Scrat 并不像 grunt 或者 gulp 那样定义不同的 task 来区分构建目的，而是通过 release 命令的多种参数组合来确定构建目的。

scrat release 命令的所有参数都可以通过 scrat release -h 命令来查看，其中常见的包括：

- --dest <paths> —— 指定构建结果的发布路径。
- --md5 —— 是否给非模块化资源添加 MD5 戳。
- --domains —— 是否给静态资源添加域名。
- --lint —— 是否开启代码校验。
- --optimize —— 是否开启代码压缩。
- --watch —— 是否开启文件监听。
- --live —— 是否开启浏览器自动刷新。

scrat release 命令的所有参数均可自由组合，通过不同参数的不同组合即可得到不同的开发状态，参数顺序没有影响。

本项目执行 scrat release 命令后，执行 scrat server open 命令，可以打开本地调试目录，如图 17-14 所示。

```
bogon:vue-scrat-demo huangyi$ scrat server open

 [NOTIC] browse /Users/huangyi/.scrat-tmp/www

bogon:vue-scrat-demo huangyi$
```

图17-14　Scrat打开本地调试目录

编译后生成的调试目录结构如下，我们重点看一下 public 目录。

```
|- www
  |-server （后端代码逻辑）
  |-public （生成的静态资源目录）
    |- c
      |- scrat-team-fastclick
        |-1.0.2
          |-fastclick.js
      |- scrat-vue
        |-1.0.0
          |- cancel
          |- common
          |- expired
          |- header
    |- home
    |- map
          |- order
        |- order.css
        |- order.css.js
        |- order.js
      |- scrat-vue
        |- 1.0.0
    |- lib
          |- index.html
    |-views   （模板目录）
    |- …
```

我们可以看到，编译后的静态资源分为 c 目录（模块化资源目录）和 scrat-vue 项目目录。c 目录又按项目名称和版本号划分为不同的子目录，这种部署规则可以保证不同的项目、不同的版本都可以增量发布而不会有任何冲突。在生产环境中，我们也可以把 public 目录的所有资源上传到 CDN 的回源机器，同时修改 fis-conf.js 中的静态资源域名，在编译过程中利用 scrat release 的 domains 参数在静态资源 URL 前添加域名。我们还可以通过修改 fis-conf.js 的配置来实现更多的功能。这块内容可以参阅 FIS 官网（http://fis.baidu.com/），这里就不再赘述了。

项目编译后，可以执行 scrat server start 命令开启本地 node 服务器，如图 17-15 所示。

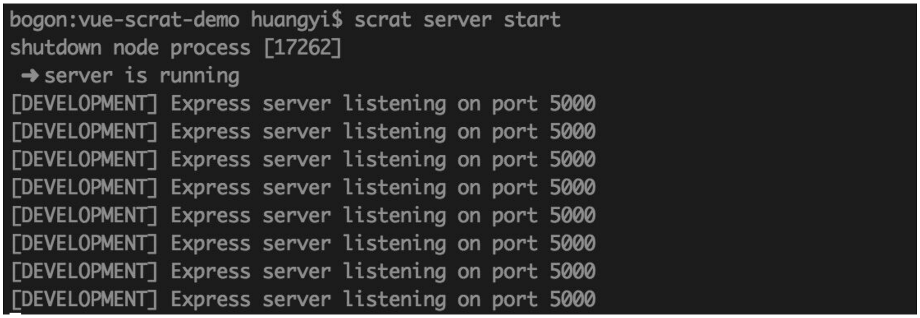


图17-15　Scrat开启本地node服务器

最后，在浏览器中输入 localhost:5000，可以预览效果，至此，整个项目的开发和部署完成。

17.6　总结

Vue.js 的设计思想是专注而灵活，它只聚焦视图层，响应的数据绑定和组件系统是其特色。当我们采用 Vue.js 构建一些大型应用时，就不得不考虑 Web 前端工程化的一些事情了，而 Scrat 在工程化方面做得特别棒。Scrat 和 Vue.js 在组件化开发思想上不谋而合，相得益彰，Scrat+Vue.js 产生的化学反应是美妙的，它们必将成为开发大型项目的利器。

第 18 章 Vue.js 2.0

Vue.js 2.0 preview 版本于 2016 年 4 月底发布，相对于 1.x 版本，Vue.js 2.0 做了不少的改进和优化。除了部分 API 变更、模板更加灵活之外，最吸引人的恐怕就是高性能的 Virtual DOM 和流式服务端渲染功能了。接下来就让我们一起走进 Vue.js 2.0 的世界，看一看 Vue.js 2.0 给我们带来了哪些惊喜。

18.1 API 变更

对于所有框架/库做大版本的更新，开发者最关心的就是 API 与之前版本的兼容程度。虽然 Vue.js 2.0 用 ES 6+flow 对整个项目进行了重写，但作者表示除了一些有意废弃掉的功能，API 和 1.x 是大部分兼容的。部分功能的废弃，本质上是为了提供更简洁的 API 而减少开发者对框架的理解和学习成本，从而提高开发者的效率。下面让我们一起来看看哪些 API 发生了变化。

18.1.1 全局配置

- 新增 Vue.config.errorHandler

Vue.js 2.0 新增了 config.errorHandler，它是一个全局钩子函数。当组件渲染时遇到未处理的异常，会调用这个函数，默认会输出错误堆栈信息。

- 新增 Vue.config.keyCodes

Vue.js 2.0 新增了 config.keyCodes，它提供了对 v-on 指令按键修饰符自定义 key 别名的配置。

- 废弃 Vue.config.debug

在 Vue.js 1.x 版本中，如果 config.debug 设置成 true，则在开发版本中为所有的警告打印栈

追踪信息。Vue.js 2.0 新增了 errorHandler 全局钩子函数，可以通过它输出错误堆栈信息，所以 config.debug 被废弃。

- 废弃 Vue.config.async

在 Vue.js 1.x 版本中，config.async 表示异步模式，默认值为 true，如果关闭异步模式，Vue 检测到数据变化时会同步更新 DOM。这样虽然方便了调试，但会导致性能下降，并影响 Watcher 的回调顺序，在生产环境下是不建议使用的。Vue.js 2.0 采用了 Virtual DOM 的方式，它的渲染方式必须是异步的，因此废弃了 config.async 这个配置。

- 废弃 Vue.config.delimeters

在 Vue.js 1.x 版本中，config.delimeters 代表文本插值的界定符，是一个全局配置。而在 Vue.js 2.0 中，该配置从全局配置中移除，变成一个组件级别的配置。

- 废弃 Vue.config.unsafeDelimeters

在 Vue.js 1.x 版本中，config.unsafeDelimeters 代表原生 HTML 插值的界定符。Vue.js 2.0 废弃了该配置，用 v-html 指令替代该功能。

18.1.2 全局 API

- 新增 Vue.compile

Vue.js 2.0 新增了 Vue.compile 全局 API，它的功能是把一个 Vue 实例的模板字符串编译成 render function。

- 废弃 Vue.transition 的 stagger

在 Vue.js 1.x 版本中，当 transition 和 v-for 一起使用时可以通过 stagger 创建渐进过渡效果。Vue.js 2.0 废弃了该接口，可以在遍历时给每个元素设置 data-index 属性，然后通过在 JavaScript 中访问该属性实现类似的效果。

- 废弃 Vue.elementDirective

在 Vue.js 1.x 版本中，Vue.elementDirective 的功能是定义全局元素指令，元素指令的本意是像 AngularJS 的 E 指令那样定义一个自定义元素，它可以被看成是一个轻量组件。但是，这是一个鸡肋功能，Vue.js 本身就有一套很好的组件化开发机制（这点和 AngularJS 不同），所以 Vue.js

2.0 废弃了该 API，可以通过 component 来实现。

- 废弃 Vue.partial

在 Vue.js 1.x 版本中，Vue.partial 的功能是注册或获取全局的 partial。partial 是一种特殊元素，它可动态地插入一小段模板，在插入时被 Vue 编译。但由于它的功能完全可以通过 functional 的 component 实现，因此在 Vue.js 2.0 中被废弃。

18.1.3 VM 选项

1. 数据部分

- 废弃 props.coorce

在 Vue.js 1.x 版本中，props.coorce 是属性转换函数，它的功能是把传入的属性值通过一个函数进行转换。不过它的功能也可以通过设置计算属性来实现，因此在 Vue.js 2.0 中被废弃。

- 废弃 prop binding modes

在 Vue.js 1.x 版本中，我们可以使用.sync、.once 绑定修饰符设置属性的绑定模式为双向或者单次绑定。Vue.js 2.0 废弃了这些模式，属性只能够单向传递，如果子组件想影响父组件，则必须通过事件派发的方法，这样做的好处是降低了子组件和父组件的耦合。在 Vue.js 2.0 中，我们应该把 prop 看作是 immutable 的，不建议直接修改 prop，所有需要修改 prop 的场景都可以用基于该 prop 值的 data 属性或者 computed 属性替代。

2. DOM 部分

- 新增 render

Vue.js 2.0 新增了 render 字段，render 字段是一个 function，运行时会把模板中的内容经过 Vue 的编译，转换成渲染方法存入 render 字段，然后再执行。如果发现 render 字段已经存在，则跳过模板解析过程直接渲染。因此，在 Vue.js 2.0 中写一个模板和写一个 render function 是等价的。

- 废弃 replace

在 Vue.js 1.x 版本中，我们可以通过 replace 属性来决定是否用模板替换挂载元素，默认值是 true，模板将覆盖挂载元素；Vue.js 2.0 废弃了该属性并且规定组件必须有确切的根元素，模板将挂载到这个根元素上并覆盖。因此，在 Vue.js 2.0 中不能把 Vue 实例挂载在 document.body 上。

3. 资源部分

- 废弃 partials

在 Vue.js 1.x 版本中，partials 是一个包含模板字符串的对象。它的功能可以通过 components 对象实现，因此在 Vue.js 2.0 中被废弃。

- 废弃 elementDirective

在 Vue.js 1.x 版本中，elementDirective 是一个包含元素指令的对象。它的功能可以通过 functional 的 components 实现，因此在 Vue.js 2.0 中被废弃。

4. 杂项部分

- 新增 delimiters

在 Vue.js 1.x 版本中，delimiters 是文本插值的界定符，它是一个全局的配置选项。在 Vue.js 2.0 中，delimiters 作为 Vue 实例的一个属性，只独立作用于当前 Vue 实例的编译过程。

- 新增 functional

在 Vue.js 1.x 版本中，我们可以通过 partial 这种特殊元素创建一个模板，它通常是无状态的。Vue.js 2.0 废弃了 partial，我们可以创建一个 functional 的 component 来替代它，仅仅通过一个 render function 来创建 Virtual DOM。

- 废弃 events

在 Vue.js 1.x 版本中，组件的自定义事件监听都是通过这个属性设置的。在 Vue.js 2.0 中，组件的事件派发和广播被废弃，因为考虑到组件树可能层级很深、兄弟组件的通信问题。我们可以通过全局事件管理中心来解决这些问题。

5. 生命期钩子函数部分

在 Vue.js 2.0 中，Vue 实例的生命周期发生了一些变化，如图 18-1 所示。

- 新增 beforeMount

Vue.js 2.0 新增了 beforeMount 钩子函数，它的调用时机是在模板编译成 render 方法之后、创建 Watcher 之前。

- 新增 mounted

Vue.js 2.0 新增了 mounted 钩子函数，它的调用时机是在 DOM 树生成之后。

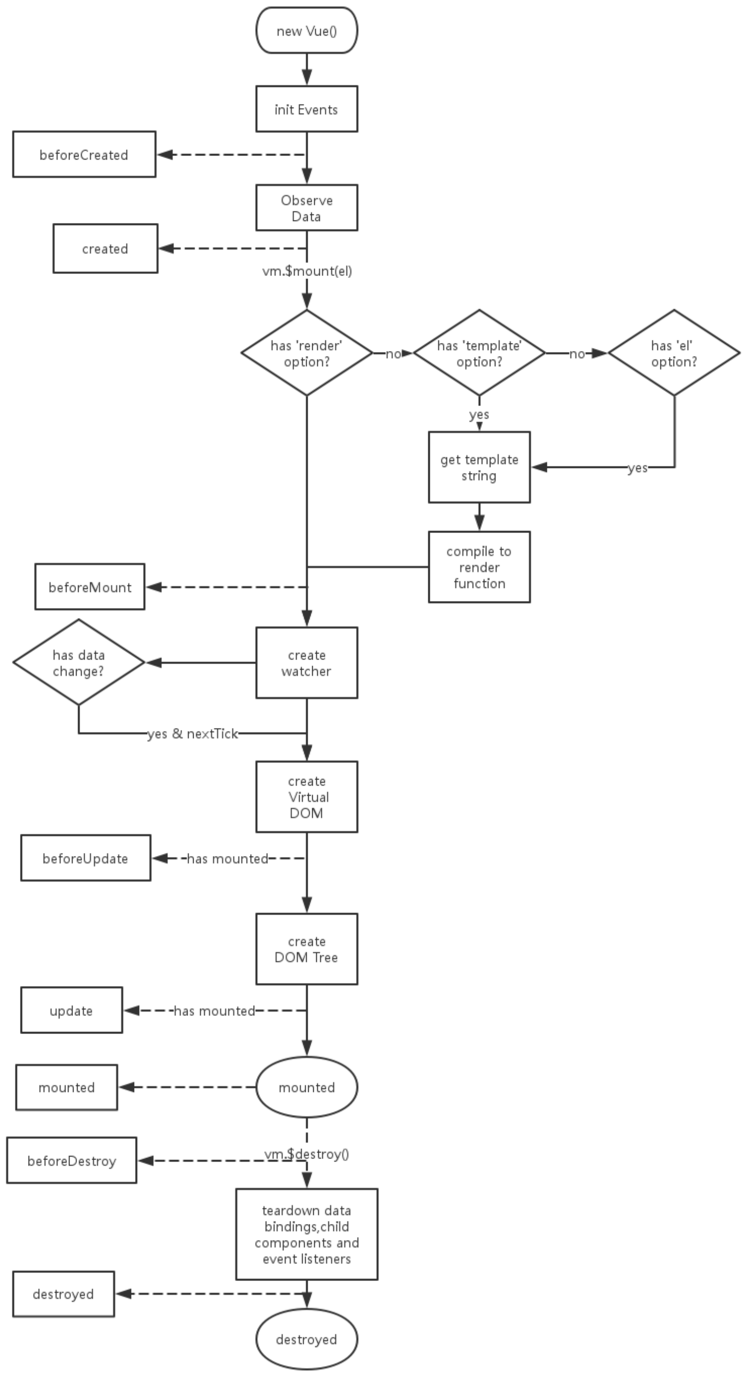


图18-1　Vue.js 2.0生命周期图示

- 新增 beforeUpdate

Vue.js 2.0 新增了 beforeUpdate 钩子函数，它的调用时机是在 Virtual DOM 生成之后、DOM 树生成之前，调用条件是这个 vm 实例已经 mounted 过。

- 新增 update

Vue.js 2.0 新增了 update 钩子函数，它的调用时机是在 DOM 树生成之后，调用条件是这个 vm 实例已经 mounted 过。

- 新增 activated

Vue.js 2.0 新增了 activated 钩子函数，它的调用时机是在 DOM 树生成之后，调用条件是 keep-alive 组件。

- 新增 deactivated

Vue.js 2.0 新增了 deactivated 钩子函数，它的调用时机是在 Vue 实例销毁时，调用条件是 keep-alive 组件。

- 废弃 ready

在 Vue.js 1.x 版本中，ready 钩子函数的调用时机是第一次插入 DOM 后。Vue.js 2.0 并不一定执行在浏览器环境中，也可能是在服务端渲染，因此废弃了该钩子函数并用 mounted 钩子函数替代。

- 废弃 beforeCompile

在 Vue.js 1.x 版本中，beforeCompile 钩子函数的调用时机是在模板编译前。Vue.js 2.0 废弃了该钩子函数并用 created 钩子函数替代。

- 废弃 compiled

在 Vue.js 1.x 版本中 compiled 钩子函数的调用时机是在编译模板之后、DOM 创建之前。Vue.js 2.0 废弃了该钩子函数并用 mounted 钩子函数替代。

- 废弃 attached

在 Vue.js 1.x 版本中 attached 钩子函数的调用时机是插入 DOM 时。Vue.js 2.0 不一定会创建真实的 DOM，因此废弃了该钩子。

❍ 废弃 detached

在 Vue.js 1.x 版本中，attached 钩子函数的调用时机是移除 DOM 时，废弃理由同上。

18.1.4　实例属性

❍ **废弃** vm.$els

在 Vue.js 1.x 版本中，vm.$els 表示一个注册有 v-el DOM 元素的对象，可以通过它访问到组件中的 DOM 对象。Vue.js 2.0 废弃了该属性，把该功能合并到 vm.$refs 中。

18.1.5　实例方法

1. 数据相关

❍ 废弃 vm.$get

在 Vue.js 1.x 版本中，vm.$get 的功能是从 Vue 实例获取指定的表达式的值。Vue.js 2.0 废弃了此方法，建议直接从 Javascript 对象中取值。

❍ 废弃 Vue.set

在 Vue.js 1.x 版本中，vm.$set 的功能是设置 Vue 实例的属性值。Vue.js 2.0 废弃了此方法，可以用全局 API Vue.set 替代。

❍ 废弃 Vue.delete

在 Vue.js 1.x 版本中，vm.$delete 的功能是删除 Vue 实例（以及$data）上的顶级属性。Vue.js 2.0 废弃了此方法，可以用全局 API Vue.delete 替代。

❍ 废弃 vm.$eval

在 Vue.js 1.x 版本中，vm.$eval 的功能是计算当前实例上的合法绑定表达式，可包含过滤器，该功能也可以通过 computed 属性实现，实际上很少使用。在 Vue.js 2.0 中被废弃。

❍ 废弃 vm.$interpolate

在 Vue.js 1.x 版本中，vm.$interpolate 的功能是计算模板，该功能也可以使用 computed 属性实现，实际上很少使用。在 Vue.js 2.0 中被废弃。

- 废弃 vm.$log

在 Vue.js 1.x 版本中，vm.$log 的功能是打印当前实例数据。Vue.js 2.0 废弃了该方法，因为它完全可以被 Vue 提供的开发者工具替代。

2. 事件相关

- 废弃 vm.$dispatch

在 Vue.js 1.x 版本中，vm.$dispatch 是组件派发事件的方法，它会沿着父链向上冒泡在触发第一个监听器后停止，除非返回 true。Vue.js 2.0 废弃了该方法，用全局的事件或者 Vuex 替代。

- 废弃 vm.$broadcast

在 Vue.js 1.x 版本中，vm.$broadcast 是组件广播事件的方法，它会向所有子组件广播，每条路径上的通知在触发一个监听器后停止，除非返回 true。Vue.js 2.0 废弃了该方法，用全局的事件或者 Vuex 替代。

3. DOM 相关

- 废弃 vm.$appendTo

在 Vue.js1.x 版本中，vm.$appendTo 的功能是将实例的 DOM 片段插入目标元素内。Vue.js 2.0 废弃了该方法，可以通过原生 DOM API: appendChild 方法配合 vm.$el 实现。

- 废弃 vm.$before

在 Vue.js1.x 版本中，vm.$before 的功能是将实例的 DOM 片段插入目标元素的前面。Vue.js 2.0 废弃了该方法，可以通过原生 DOM API: insertBefore 方法配合 vm.$el 实现。

- 废弃 vm.$after

在 Vue.js1.x 版本中，vm.$after 的功能是将实例的 DOM 片段插入目标元素后面。Vue.js 2.0 废弃了该方法，可以通过原生 DOM API: afterChild 方法和 appendChild 方法配合 vm.$el 实现。

- 废弃 vm.$remove

在 Vue.js1.x 版本中，vm.$remove 的功能从 DOM 中删除实例的 DOM 元素或者片段。Vue.js 2.0 废弃了该方法，可以通过原生 DOM API: remove 方法配合 vm.$el 实现。

18.1.6 指令

- 新增 v-once 指令

在 Vue.js 2.0 中，可以给 DOM 元素设置 v-once 指令，来表明该元素是一个静态根节点，这些节点生成后内容就不会被改变，因此在 Virtual DOM 的 diff 和 patch 的过程中，可以忽略这些节点来提升性能。

- 废弃 v-mode 中的 debounce 参数

在 Vue.js 1.x 版本中，v-model 指令主要用来在表单控件元素上创建双向数据绑定，它支持 debounce 参数，作用是针对 input 事件设置一个最小延时，在每次敲击之后延迟同步输入框的值与数据。注意，debounce 参数并不会延迟 input 事件，它是延迟写入底层数据，因此在使用 debounce 时应当用 vm.$watch()响应数据变化。由于延迟写入底层数据，它的状态更新也会被延迟，这样就会带来一些限制，比如不能实时检测到数据的输入。Vue.js 2.0 废弃了该参数，把延迟操作和 Vue 本身解耦，可以通过 v-on:input 绑定 input 事件，并配合第三方的 debounce 方法，这样既可以实时更新状态，也可以达到延迟 input 事件的目的。

- 废弃 v-ref 指令

在 Vue.js 1.x 版本中，v-ref 指令是在父组件上注册一个子组件的索引，便于直接访问。Vue.js 2.0 废弃了这个指令，它的功能用特殊的属性 ref 替代。

- 废弃 v-el 指令

在 Vue.js 1.x 版本中，v-el 指令是为 DOM 元素注册一个索引，方便通过所属实例的$els 访问这个元素。Vue.js 2.0 废弃了这个指令，它的功能合并到 ref 属性中。

- v-for 指令修改

在 Vue.js 1.x 版本中，可以使用 v-for 指令基于数组渲染一个列表。这个指令使用特殊的语法，形式为 item in items，items 是数据数组，item 是当前数组元素的值。在 Vue.js 2.0 中，不仅可以遍历数组，也可以遍历对象。数组的遍历语法形式为 value in arr 或者(value, index)in arr，value 为当前数组元素的值，index 为当前数组元素的索引。对象的遍历语法形式为 value in obj、(value, key)in obj 或者(value, key, index)in obj。其中 value 为当前对象元素的值，key 为对象的属性名称，index 为当前遍历的索引。因此，Vue.js 2.0 废弃了 v-for 指令的$index 和$key 语法。在 Vue.js 1.x 版本中，可以通过 track-by 属性优化 v-for，让 Vue.js 尽可能复用已有的实例。在 Vue.js

2.0 中，track-by 属性用 key 属性代替，它的值可以是一个普通字符串，也可以通过 v-bind 动态绑定。

18.1.7　特殊元素

- 废弃 partial 元素

在 Vue.js 1.x 版本中，partial 元素的作用是动态插入一个模板。Vue.js 2.0 废弃了该元素，它的功能可以通过 functional 的 component 实现。

18.1.8　服务端渲染

服务端渲染是 Vue.js 2.0 的新特性，它提供的接口如下：

- 新增 renderToString(component,done)方法

它的功能是在服务端把 Vue 组件 component 渲染成 DOM 字符串，我们可以在 done 的回调函数中拿到渲染好的 DOM 字符串。

- 新增 renderToStream(component)方法

它的功能是返回一个渲染流，是一个可读的 stream，可以直接 pipe 到 HTTP Response 中。该方法相比于 renderToString，当 App 非常复杂时也不会阻塞服务器的 event loop，能够确保服务端的响应度，也能让用户更快地获得渲染内容。

18.2　Virtual DOM

18.2.1　认识 Virtual DOM

现在简单回顾一下网页开发的历程。传统的 Web 页面都是简单的静态页面加上一些基本的 DOM 操作，我们可以使用一些 JavasScript 库如 jQuery 来方便开发。随着 Web 应用程序越来越复杂，程序状态越来越多，DOM 操作的复杂度也在提升，jQuery 已经不能满足我们的开发需求了，于是一些 MVC、MVVM 框架被引入。在 MVVM 框架中，我们可以通过数据绑定实现视图和模型（数据）的互动效果，通过双向绑定自动实现模型与视图的同步更新。但这些仅仅是从代码组织方式来降低维护这种复杂应用的难度，并没有从本质上减少所维护的状态，也没有减少页面的 DOM 操作（框架替我们做了而已）。

MVVM 框架可以很好地降低我们维护状态—视图的复杂程度。但是当页面变得非常复杂时，视图的更新也可能会引发大量的 DOM 操作，产生一定的性能问题。DOM 是很“昂贵”的，我们可以把一个简单的 div 元素的属性都打印出来，如图 18-2 所示。

```
> var div = document.createElement('div')
  var str = ''
  for (var key in div){
    str += key + ' '
  }
< "align title lang translate dir dataset hidden tabIndex accessKey draggable spellcheck contentEditable isContentEditable
  offsetParent offsetTop offsetLeft offsetWidth offsetHeight style innerText outerText webkitdropzone onabort onblur
  oncancel oncanplay oncanplaythrough onchange onclick onclose oncontextmenu oncuechange ondblclick ondrag ondragend
  ondragenter ondragleave ondragover ondragstart ondrop ondurationchange onemptied onended onerror onfocus oninput
  oninvalid onkeydown onkeypress onkeyup onload onloadeddata onloadedmetadata onloadstart onmousedown onmouseenter
  onmouseleave onmousemove onmouseout onmouseover onmouseup onmousewheel onpause onplay onplaying onprogress onratechange
  onreset onresize onscroll onseeked onseeking onselect onshow onstalled onsubmit onsuspend ontimeupdate ontoggle
  onvolumechange onwaiting click focus blur onautocomplete onautocompleteerror namespaceURI prefix localName tagName id
  className classList attributes innerHTML outerHTML shadowRoot scrollTop scrollLeft scrollWidth scrollHeight clientTop
  clientLeft clientWidth clientHeight onbeforecopy onbeforecut onbeforepaste oncopy oncut onpaste onsearch onselectstart
  onwheel onwebkitfullscreenchange onwebkitfullscreenerror previousElementSibling nextElementSibling children
  firstElementChild lastElementChild childElementCount hasAttributes getAttribute getAttributeNS setAttribute
  setAttributeNS removeAttribute removeAttributeNS hasAttribute hasAttributeNS getAttributeNode getAttributeNodeNS
  setAttributeNode setAttributeNodeNS removeAttributeNode closest matches webkitMatchesSelector getElementsByTagName
  getElementsByTagNameNS getElementsByClassName insertAdjacentElement insertAdjacentText insertAdjacentHTML
  createShadowRoot getDestinationInsertionPoints requestPointerLock getClientRects getBoundingClientRect scrollIntoView
  scrollIntoViewIfNeeded animate remove webkitRequestFullScreen webkitRequestFullscreen querySelector querySelectorAll
  ELEMENT_NODE ATTRIBUTE_NODE TEXT_NODE CDATA_SECTION_NODE ENTITY_REFERENCE_NODE ENTITY_NODE PROCESSING_INSTRUCTION_NODE
  COMMENT_NODE DOCUMENT_NODE DOCUMENT_TYPE_NODE DOCUMENT_FRAGMENT_NODE NOTATION_NODE DOCUMENT_POSITION_DISCONNECTED
  DOCUMENT_POSITION_PRECEDING DOCUMENT_POSITION_FOLLOWING DOCUMENT_POSITION_CONTAINS DOCUMENT_POSITION_CONTAINED_BY
  DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC nodeType nodeName baseURI isConnected ownerDocument parentNode parentElement
  childNodes firstChild lastChild previousSibling nextSibling nodeValue textContent hasChildNodes normalize cloneNode
  isEqualNode isSameNode compareDocumentPosition contains lookupPrefix lookupNamespaceURI isDefaultNamespace insertBefore
  appendChild replaceChild removeChild addEventListener removeEventListener dispatchEvent "
```

图18-2　div元素包含的属性

可以看到，真正的 DOM 元素是非常庞大的，因为浏览器的标准就把 DOM 设计得非常复杂。更糟的是，很多时候我们在调用 DOM API 时做得不够好，导致更慢。比如有一个列表，代码示例如下：

```
<ul>
  <li>DDFE 小 Z</li>
  <li>DDFE 小 H></li>
  <li>DDFE 小 S></li>
</ul>
```

我们想把它变成这样：

```
<ul>
<li>DDFE 小 W</li>
<li>DDFE 小 J></li>
<li>DDFE 小 S></li>
<li>DDFE 小 G></li>
</ul>
```

通常的操作是把前 3 个 li 删除，再添加 4 个 li，这里面就有 3 次 Element 的删除，4 次 Element

的添加操作，这当然也是有优化空间的。

了解到 DOM 的一些弊端，人们就会思考有没有什么方式可以对 DOM 做优化。聪明的人提出了 Virtual DOM 的概念，它是一种虚拟 DOM 技术，本质上是基于 JavaScript 实现的。相对于 DOM 对象，JavaScript 对象更简单，处理速度更快，DOM 树的结构、属性信息都可以很容易地用 JavaScript 对象来表示。代码示例如下：

```
var element = {
  tagName: 'ul', // 节点标签名
  props: { // DOM 的属性，用一个对象存储键值对
    id: 'list'
  },
  children: [ // 该节点的子节点
    {tagName: 'li', props: {class: 'item'}, children: ["DDFE 小 Z"]},
    {tagName: 'li', props: {class: 'item'}, children: ["DDFE 小 H”]},
    {tagName: 'li', props: {class: 'item'}, children: ["DDFE 小 S"]}
  ]
}
```

对应的 HTML 写法是：

```
<ul id=”list”>
  <liclass=”item”>DDFE 小 Z</li>
  <li class=”item”>DDFE 小 H></li>
  <liclass=”item”>DDFE 小 S></li>
</ul>
```

我们通过 JavaScript 对象表示的树结构来构建一棵真正的 DOM 树，当数据状态发生变化时可以直接修改这个 JavaScript 对象，接着对比修改后的 JavaScript 对象，记录下需要对页面做的 DOM 操作，然后将其应用到真正的 DOM 树，实现视图的更新。这个过程就是 Virtual DOM 的核心思想。

18.2.2 Virtual DOM 在 Vue.js 2.0 中的实现

1. 创建 VNode 对象模拟 DOM 树

在 Vue.js 2.0 中，Virtual DOM 是通过 VNode 类来表达的，每一个原生 DOM 元素或者 Vue 组件都对应一个 VNode 对象。我们先来看一下 VNode 的数据结构，如图 18-3 所示。

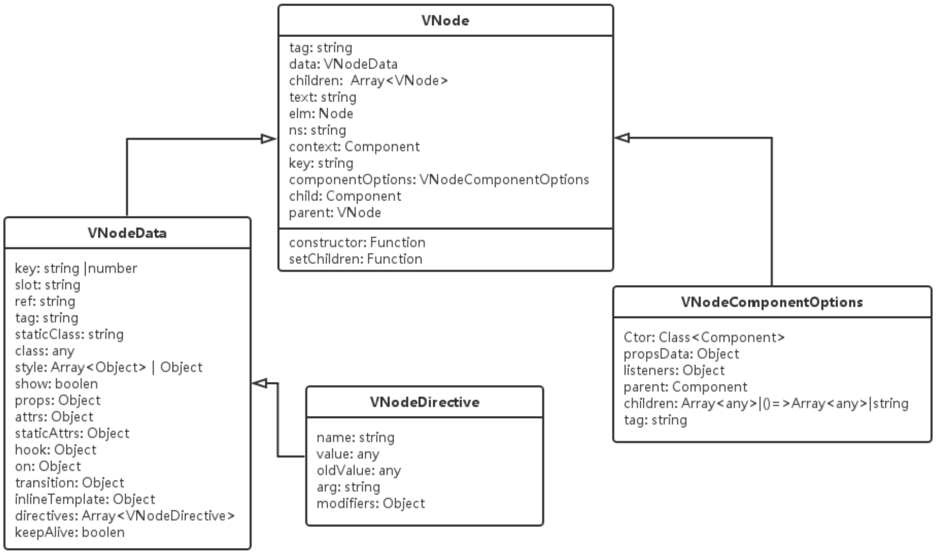


图18-3　VNode的数据结构图

图中不仅包含了 VNode 的数据结构，还包含了 VNodeData、VNodeDirective、VNodeComponentOptions 等数据结构。

（1）VNode

VNode 用来描述 DOM 节点的主要信息，包括 tag、text、elm、data、parent、children 等属性。它主要有两种生成方式，其中一种是由普通 DOM 元素生成；另一种是由 Vue 组件生成。这两种方式生成的 VNode 对象的区别主要是在 componentOptions 的值上，如果是普通 DOM 元素生成的 VNode 对象，该值为空。

（2）VNodeComponentOptions

VNode 中 componentOptions 属性的数据类型用来描述通过 Vue 组件生成 VNode 对象的一些组件相关参数，包括 Ctor、propData、listeners、parent、children、tag 等属性。

（3）VNodeData

VNode 中 data 属性的数据类型用来描述 VNode 包含的一些节点数据，包括 slot、ref、staticClass、style、class、props、attrs、transition、directives 等。

（4）VNodeDirective

VNodeData 中 directives 属性的数据类型用来描述 VNode 存储的指令数据，包括 name、

value、oldValue、arg、modifiers 等。

在了解了 VNode 及相关数据结构后，我们心中不免产生疑问，VNode 是如何在 Vue 中生成的呢？可以先通过图 18-4 看一下 VNode 的生成过程。

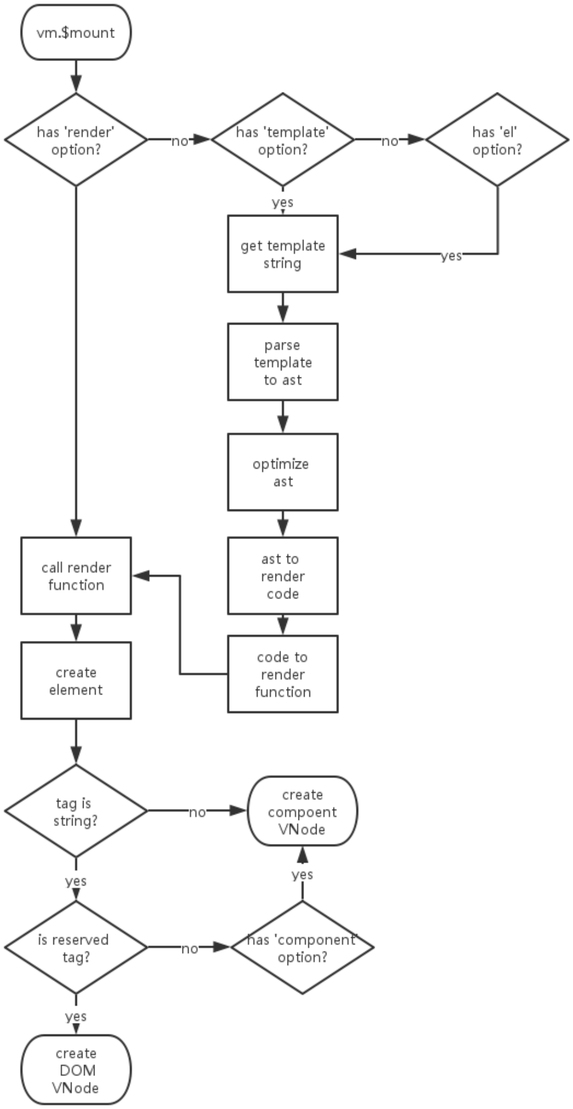


图18-4　VNode的生成过程

从图中我们可以看到，VNode 生成最关键的点是通过 call render function，render 方法在

Vue.js 2.0 中有两种生成方式：第一种是直接在 Vue 对象的 option 中添加 render 字段；第二种是像 Vue.js 1.x 版本那样写一个模板或者指定一个 el 根元素，它会首先转换成模板，经过 HTML 语法解析器生成一个 ast 抽象语法树，对语法树做优化，然后把语法树转换成代码片段，最后通过代码片段生成 function 添加到 option 的 render 字段中。

在整个过程中，特别值得一提的是 ast 语法树优化的过程，它主要做了两件事情：

- 会检测出静态的 class 名和 attributes，这样它们在初始化渲染之后就永远都不会再被比对了。
- 会检测出最大的静态子树（就是不需要动态性的子树）并且从渲染函数中萃取出来。这样在每次重渲染时，它就会直接重用完全相同的 Vnode，同时跳过比对。

接下来我们可以通过一段简单的代码看一下编译后的渲染函数。代码示例如下：

```
<div id="app">
  <h1>Hello {{who}}</h1>
</div>
<script>
  new Vue({
    el: '#app',
    data: {
      who: 'DDFE'
    }
  });
</script>
```

我们在 Vue 对象中指定了 el 根元素，经过一系列的编译操作后，最终生成 render 方法。代码示例如下：

```
(function () {
  with (this) {
    return _h(_e('div',
      {staticAttrs: {"id": "app"}}),
       [_h(_e('h1'),
         [("Hello " + _s(who))]
      )]
    )
  }
})
```

可以看到，render 方法使用了 with 方法来包裹代码快，with(this)中的属性和方法相当于通过 this 来调用，这样写是为了减少代码量。这里的 this 指向的是 Vue 对象实例，_h、_e 方法都是和 VNode 相关创建的方法。源码定义如下：

```
<!--源码目录：src/core/instance/render.js-->
Vue.prototype._h = renderElementWithChildren
Vue.prototype._e = renderElement
```

_h 和_e 方法分别是 renderElementWithChildren、renderElement，源码定义如下：

```
<!--源码目录：src/core/vdom/create-element.js-->
export function renderElementWithChildren (
  vnode: VNode | void,
  children: VNodeChildren | void
): VNode | Array<VNode> | void {
  if (vnode) {
    const componentOptions = vnode.componentOptions
    if (componentOptions) {
      if (process.env.NODE_ENV !== 'production' &&
        children && typeof children !== 'function') {
        warn(
          'A component\'s children should be a function that returns the ' +
          'children array. This allows the component to track the children ' +
          'dependencies and optimizes re-rendering.'
        )
      }
      const CtorOptions = componentOptions.Ctor.options
      // functional component
      if (CtorOptions.functional) {
        return CtorOptions.render.call(
          null,
          componentOptions.parent.$createElement, // h
          componentOptions.propsData || {},      // props
          normalizeChildren(children)           // children
        )
      } else {
        // normal component
        componentOptions.children = children
      }
    } else {
      // normal element
```

```
      vnode.setChildren(normalizeChildren(children))
    }
  }
  return vnode
}

export function renderElement (
  tag?: string | Class<Component> | Function | Object,
  data?: VNodeData,
  namespace?: string
): VNode | void {
  // make sure to expose real self instead of proxy
  const context: Component = this._self
  const parent: ?Component = renderState.activeInstance
  const host = context !== parent ? parent : undefined
  if (!parent) {
    process.env.NODE_ENV !== 'production' && warn(
      'createElement cannot be called outside of component ' +
      'render functions.'
    )
    return
  }
  if (!tag) {
    // in case of component :is set to falsy value
    return emptyVNode()
  }
  if (typeof tag === 'string') {
    let Ctor
    if (config.isReservedTag(tag)) {
      return new VNode(
        tag, data,
        undefined, undefined, undefined,
        namespace, context, host
      )
    } else if ((Ctor = resolveAsset(context.$options, 'components', tag))) {
      return createComponent(Ctor, data, parent, context, host, tag)
    } else {
      if (process.env.NODE_ENV !== 'production') {
        if (
          !namespace &&
          !(config.ignoredElements && config.ignoredElements.indexOf(tag) > -1) &&
```

```
          config.isUnknownElement(tag)
        ) {
          warn(
            'Unknown custom element: <' + tag + '> - did you ' +
            'register the component correctly? For recursive components, ' +
            'make sure to provide the "name" option.'
          )
        }
      }
      return new VNode(
        tag, data,
        undefined, undefined, undefined,
        namespace, context, host
      )
    }
  } else {
    return createComponent(tag, data, parent, context, host)
  }
}
```

renderElementWithChildren 方法的功能是给一个 VNode 对象添加若干子 VNode，因为整个 Virtual DOM 是一种树状结构，每个节点都可能会有若干子节点。

renderElement 方法的功能是创建一个 VNode 对象，如果是一个 reserved tag（比如 html、head 等一些合法的 HTML 标签），则会创建普通的 DOM VNode 对象；如果是一个 component tag（通过 Vue 注册的自定义 component），则会创建 Component VNode 对象，它的 VNodeComponentOptions 不为 null。

创建好 VNode，下一步就是要把 Virtual DOM 渲染成真正的 DOM，我们看一下 Vue.js 2.0 是怎么做的。

2. VNode patch 生成 DOM

在 Vue.js 2.0 中，VNode 转换成真正的 DOM 是通过 patch(oldVnode, vnode, hydrating)方法实现的，来看一下 patch 方法。源码定义如下：

```
<!--源码目录：src/core/vdom/patch.js-->
return function patch (oldVnode, vnode, hydrating) {
    let elm, parent
    const insertedVnodeQueue = []
    if (!oldVnode) {
```

```
      // empty mount, create new root element
      createElm(vnode, insertedVnodeQueue)
    } else {
      const isRealElement = isDef(oldVnode.nodeType)
      if (!isRealElement && sameVnode(oldVnode, vnode)) {
        patchVnode(oldVnode, vnode, insertedVnodeQueue)
      } else {
        if (isRealElement) {
          // mounting to a real element
          // check if this is server-rendered content and if we can perform
          // a successful hydration.
          if (oldVnode.hasAttribute('server-rendered')) {
            oldVnode.removeAttribute('server-rendered')
            hydrating = true
          }
          if (hydrating) {
            if (hydrate(oldVnode, vnode, insertedVnodeQueue)) {
              invokeInsertHook(insertedVnodeQueue)
              return oldVnode
            } else if (process.env.NODE_ENV !== 'production') {
              warn(
                'The client-side rendered virtual DOM tree is not matching ' +
                'server-rendered content. Bailing hydration and performing ' +
                'full client-side render.'
              )
            }
          }
          // either not server-rendered, or hydration failed.
          // create an empty node and replace it
          oldVnode = emptyNodeAt(oldVnode)
        }
        elm = oldVnode.elm
        parent = nodeOps.parentNode(elm)
        createElm(vnode, insertedVnodeQueue)
        if (parent !== null) {
          nodeOps.insertBefore(parent, vnode.elm, nodeOps.nextSibling(elm))
          removeVnodes(parent, [oldVnode], 0, 0)
        } else if (isDef(oldVnode.tag)) {
          invokeDestroyHook(oldVnode)
        }
      }
```

```
  }
  invokeInsertHook(insertedVnodeQueue)
  return vnode.elm
}
```

patch 方法支持 3 个参数，其中 oldVnode 是一个真实的 DOM 或者一个 VNode 对象，它表示当前的 VNode；vnode 是 VNode 对象类型，它表示待替换的 VNode；hydrating 是 bool 类型，它表示是否直接使用服务端渲染的 DOM 元素，因为 Vue.js 2.0 也支持服务端渲染。我们可以通过流程图看一下 patch 方法的运行逻辑，如图 18-5 所示。

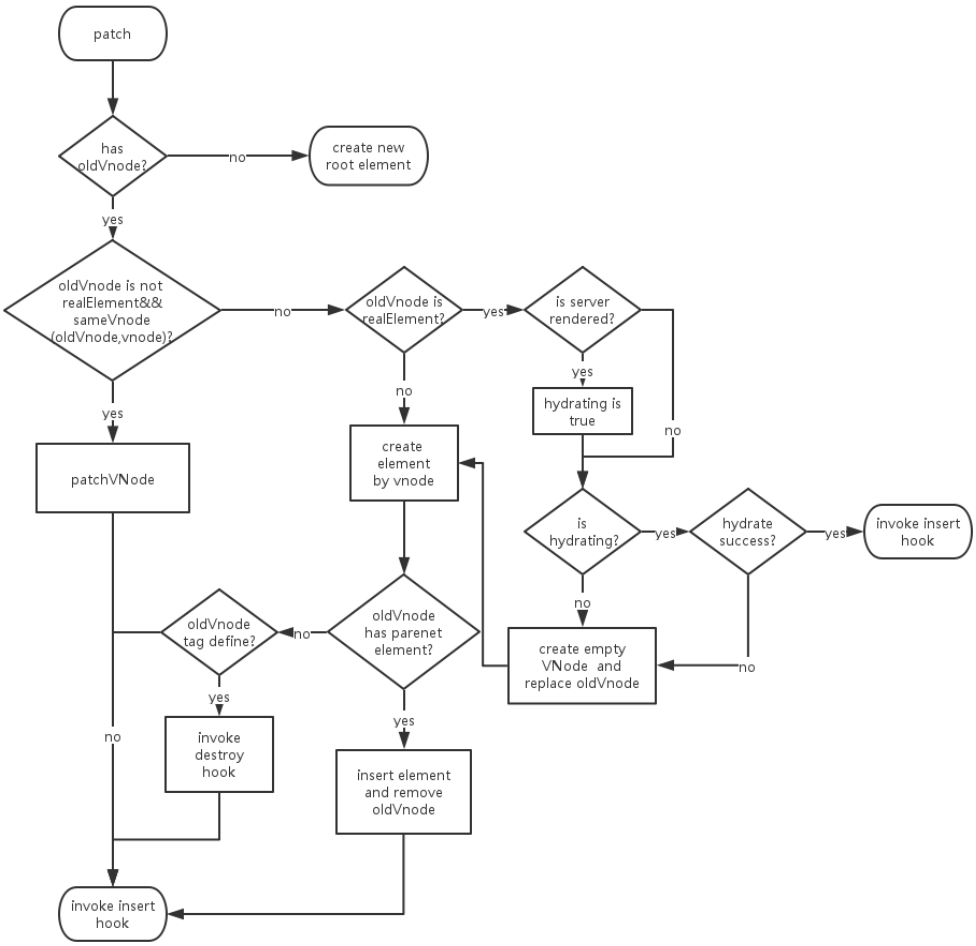


图18-5　patch方法运行逻辑图

patch 方法的运行逻辑看上去比较复杂，其中有两个方法 createElm 和 patchVnode 是生成

DOM 的关键。源码定义如下：

```
<!--源码目录：src/core/vdom/patch.js-->
function createElm (vnode, insertedVnodeQueue) {
    let i, elm
    const data = vnode.data
    if (isDef(data)) {
      if (isDef(i = data.hook) && isDef(i = i.init)) i(vnode)
      // after calling the init hook, if the vnode is a child component
      // it should've created a child instance and mounted it. the child
      // component also has set the placeholder vnode's elm.
      // in that case we can just return the element and be done.
      if (isDef(i = vnode.child)) {
        invokeCreateHooks(vnode, insertedVnodeQueue)
        setScope(vnode)
        return vnode.elm
      }
    }
    const children = vnode.children
    const tag = vnode.tag
    if (isDef(tag)) {
      elm = vnode.elm = vnode.ns
        ? nodeOps.createElementNS(vnode.ns, tag)
        : nodeOps.createElement(tag)
      setScope(vnode)
      if (Array.isArray(children)) {
        for (i = 0; i < children.length; ++i) {
          nodeOps.appendChild(elm, createElm(children[i], insertedVnodeQueue))
        }
      } else if (isPrimitive(vnode.text)) {
        nodeOps.appendChild(elm, nodeOps.createTextNode(vnode.text))
      }
      if (isDef(data)) {
        invokeCreateHooks(vnode, insertedVnodeQueue)
      }
    } else {
      elm = vnode.elm = nodeOps.createTextNode(vnode.text)
    }
    return vnode.elm
  }
```

```
function patchVnode (oldVnode, vnode, insertedVnodeQueue) {
    if (oldVnode === vnode) return
    let i, hook
    if (isDef(i = vnode.data) && isDef(hook = i.hook) && isDef(i = hook.prepatch)) {
      i(oldVnode, vnode)
    }
    const elm = vnode.elm = oldVnode.elm
    const oldCh = oldVnode.children
    const ch = vnode.children
    if (isDef(vnode.data)) {
      for (i = 0; i < cbs.update.length; ++i) cbs.update[i](oldVnode, vnode)
      if (isDef(hook) && isDef(i = hook.update)) i(oldVnode, vnode)
    }
    if (isUndef(vnode.text)) {
      if (isDef(oldCh) && isDef(ch)) {
        if (oldCh !== ch) updateChildren(elm, oldCh, ch, insertedVnodeQueue)
      } else if (isDef(ch)) {
        if (isDef(oldVnode.text)) nodeOps.setTextContent(elm, '')
        addVnodes(elm, null, ch, 0, ch.length - 1, insertedVnodeQueue)
      } else if (isDef(oldCh)) {
        removeVnodes(elm, oldCh, 0, oldCh.length - 1)
      } else if (isDef(oldVnode.text)) {
        nodeOps.setTextContent(elm, '')
      }
    } else if (oldVnode.text !== vnode.text) {
      nodeOps.setTextContent(elm, vnode.text)
    }
    if (isDef(vnode.data)) {
      for (i = 0; i < cbs.postpatch.length; ++i) cbs.postpatch[i](oldVnode, vnode)
      if (isDef(hook) && isDef(i = hook.postpatch)) i(oldVnode, vnode)
    }
  }
```

（1）createElm(vnode, insertedVnodeQueue)

该方法会根据 vnode 的数据结构创建真实的 DOM 节点，如果 vnode 有 children，则会遍历这些子节点，递归调用 createElm 方法。InsertedVnodeQueue 是记录子节点创建顺序的队列，每创建一个 DOM 元素就会往这个队列中插入当前的 VNode。当整个 VNode 对象全部转换成为真实的 DOM 树时，会依次调用这个队列中 VNode hook 的 insert 方法。

（2）patchVnode(oldVnode, vnode, insertedVnodeQueue)

该方法会通过比较新旧 VNode 节点，根据不同的状态对 DOM 做合理的更新操作（添加、移动、删除等），整个过程还会依次调用 prepatch、update、postpatch 等钩子函数。在编译阶段生成的一些静态子树，在这个过程中由于不会改变而直接跳过比对。动态子树在比较过程中比较核心的部分就是当新旧 VNode 同时存在 children，通过 updateChildren 方法对子节点做更新。Vue.js 2.0 在这块的实现很精彩，接下来我们重点看一下这个方法的实现。源码定义如下：

```
<!--源码目录：src/core/vdom/patch.js-->
function updateChildren (parentElm, oldCh, newCh, insertedVnodeQueue) {
    let oldStartIdx = 0
    let newStartIdx = 0
    let oldEndIdx = oldCh.length - 1
    let oldStartVnode = oldCh[0]
    let oldEndVnode = oldCh[oldEndIdx]
    let newEndIdx = newCh.length - 1
    let newStartVnode = newCh[0]
    let newEndVnode = newCh[newEndIdx]
    let oldKeyToIdx, idxInOld, elmToMove, before

    while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
      if (isUndef(oldStartVnode)) {
        oldStartVnode = oldCh[++oldStartIdx] // Vnode has been moved left
      } else if (isUndef(oldEndVnode)) {
        oldEndVnode = oldCh[--oldEndIdx]
      } else if (sameVnode(oldStartVnode, newStartVnode)) {
        patchVnode(oldStartVnode, newStartVnode, insertedVnodeQueue)
        oldStartVnode = oldCh[++oldStartIdx]
        newStartVnode = newCh[++newStartIdx]
      } else if (sameVnode(oldEndVnode, newEndVnode)) {
        patchVnode(oldEndVnode, newEndVnode, insertedVnodeQueue)
        oldEndVnode = oldCh[--oldEndIdx]
        newEndVnode = newCh[--newEndIdx]
      } else if (sameVnode(oldStartVnode, newEndVnode)) { // Vnode moved right
        patchVnode(oldStartVnode, newEndVnode, insertedVnodeQueue)
        nodeOps.insertBefore(parentElm, oldStartVnode.elm, nodeOps.nextSibling(oldEndVnode.elm))
        oldStartVnode = oldCh[++oldStartIdx]
        newEndVnode = newCh[--newEndIdx]
      } else if (sameVnode(oldEndVnode, newStartVnode)) { // Vnode moved left
        patchVnode(oldEndVnode, newStartVnode, insertedVnodeQueue)
```

```
      nodeOps.insertBefore(parentElm, oldEndVnode.elm, oldStartVnode.elm)
      oldEndVnode = oldCh[--oldEndIdx]
      newStartVnode = newCh[++newStartIdx]
    } else {
      if (isUndef(oldKeyToIdx)) oldKeyToIdx = createKeyToOldIdx(oldCh, oldStartIdx,
oldEndIdx)
      idxInOld = oldKeyToIdx[newStartVnode.key]
      if (isUndef(idxInOld)) { // New element
        nodeOps.insertBefore(parentElm, createElm(newStartVnode, insertedVnodeQueue),
oldStartVnode.elm)
        newStartVnode = newCh[++newStartIdx]
      } else {
        elmToMove = oldCh[idxInOld]
        /* istanbul ignore if */
        if (process.env.NODE_ENV !== 'production' && !elmToMove) {
          warn(
            'It seems there are duplicate keys that is causing an update error. ' +
            'Make sure each v-for item has a unique key.'
          )
        }
        if (elmToMove.tag !== newStartVnode.tag) {
          // same key but different element. treat as new element
          nodeOps.insertBefore(parentElm, createElm(newStartVnode, insertedVnodeQueue),
oldStartVnode.elm)
          newStartVnode = newCh[++newStartIdx]
        } else {
          patchVnode(elmToMove, newStartVnode, insertedVnodeQueue)
          oldCh[idxInOld] = undefined
          nodeOps.insertBefore(parentElm, newStartVnode.elm, oldStartVnode.elm)
          newStartVnode = newCh[++newStartIdx]
        }
      }
    }
  }
  if (oldStartIdx > oldEndIdx) {
    before = isUndef(newCh[newEndIdx + 1]) ? null : newCh[newEndIdx + 1].elm
    addVnodes(parentElm, before, newCh, newStartIdx, newEndIdx, insertedVnodeQueue)
  } else if (newStartIdx > newEndIdx) {
    removeVnodes(parentElm, oldCh, oldStartIdx, oldEndIdx)
  }
}
```

updateChildren 这个方法看上去比较复杂，它主要通过 while 循环一遍遍对比两棵树的子节点来更新 DOM。为了更加直观地理解整个比较更新过程，我们可以通过一个例子来模拟一下。

假设有新旧两棵树，树中的子节点分别用 a、b、c、d 等代号表示，不同的代号代表不同的 VNode。初始状态如图 18-6 所示。

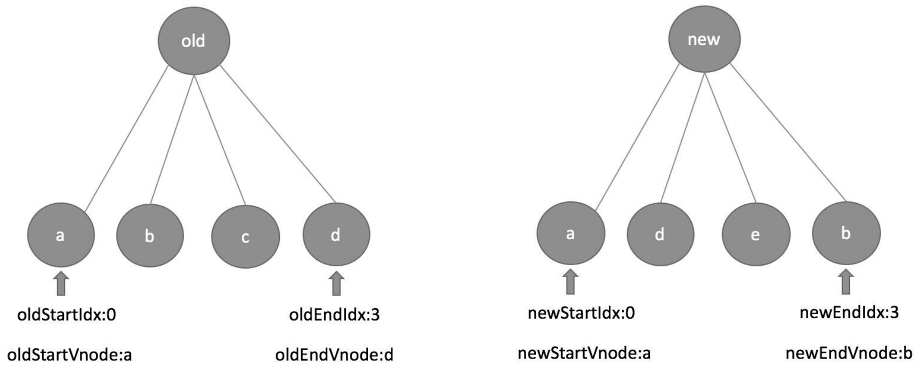


图18-6　新旧VNode子节点更新——初始状态

在设置好初始状态后，我们开始第一遍比较。此时 oldStartVnode 为 a，newStartVnode 也为 a，命中了 sameVnode(oldStartVnode, newStartVnode)逻辑，则直接调用 patchVnode (oldStartVnode, newStartVnode, insertedVnodeQueue)方法更新节点 a。接着把 oldStartIdx 和 newStartIdx 索引分别 +1，形成状态如图 18-7 所示。

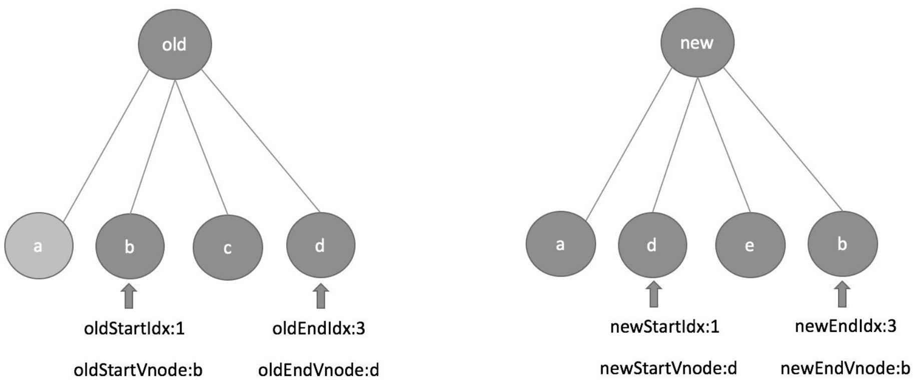


图18-7　新旧VNode子节点更新——step1

更新完节点 a 后，我们开始第二遍比较。此时 oldStartVnode 为 b，newEndVnode 也为 b，命中了 sameVnode(oldStartVnode, newEndVnode)逻辑，则调用 patchVnode(oldStartVnode,

newEndVnode, insertedVnodeQueue)方法更新节点 b。接着调用 nodeOps.insertBefore(parentElm, oldStartVnode.elm, nodeOps.nextSibling(oldEndVnode.elm))方法把节点 b 移到树的最右边。最后把 oldStartIdx 索引+1，newEndIdx 索引-1，形成状态如 18-8 所示。

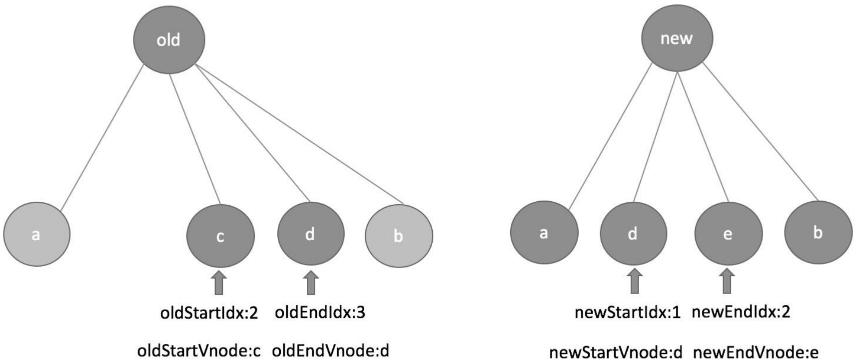


图18-8 新旧VNode子节点更新——step2

更新完节点 b 后，我们开始第三遍比较。此时 oldEndVnode 为 d，newStartVnode 也为 d，命中了 sameVnode(oldEndVnode, newStartVnode)逻辑，则调用 patchVnode(oldEndVnode, newStartVnode, insertedVnodeQueue)方法更新节点 d。接着调用 nodeOps.insertBefore(parentElm, oldEndVnode.elm, oldStartVnode.elm)方法把节点 d 移到节点 c 的左边。最后把 oldEndIdx 索引-1，newStartIdx 索引+1，形成状态如 18-9 所示。

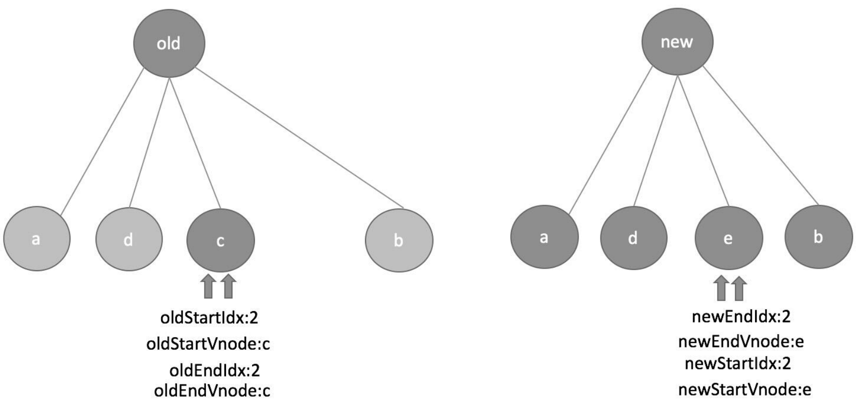


图18-9 新旧VNode子节点更新——step3

更新完节点 d 后，我们开始第四遍比较。此时 newStartVnode 为 e，节点 e 在旧树里是没有的，因此应该被作为一个新元素插入，调用 nodeOps.insertBefore(parentElm, createElm

(newStartVnode, insertedVnodeQueue), oldStartVnode.elm)方法把节点 e 插入到节点 c 之前。接着把 newStartIdx 索引+1，形成状态如 18-10 所示。

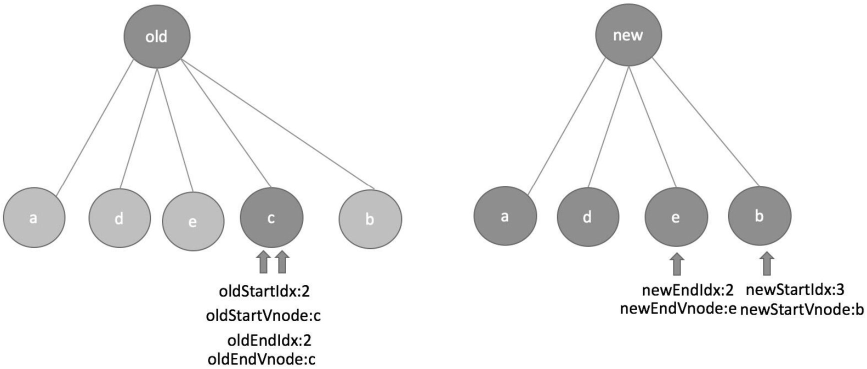


图18-10　新旧VNode子节点更新——step4

插入节点 e 后，我们可以看到 newStartIdex 已经大于 newEndIdx 了，while 循环已经完毕。接着调用 removeVnodes(parentElm, oldCh, oldStartIdx, oldEndIdx)删除旧树中剩余的节点 c，最终形成状态如 18-11 所示。

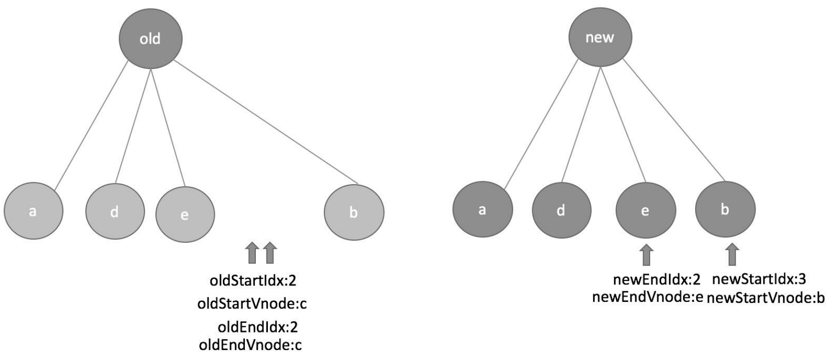


图18-11　新旧VNode子节点更新——更新完成

updateChildren 通过以上几步操作完成了旧树子节点的更新，可以看到实际上只用了一些比较小的 DOM 操作，比起用 innerHTML 直接替换所有子节点这种方法，在性能上有所提升，并且当子节点越复杂时，这种提升效果就越明显。

VNode 通过 patch 方法生成 DOM 后，会调用 mounted hook。至此，整个 Vue 实例就创建完成了。当这个 Vue 实例的 Watcher 观察到数据变化时，会再次调用 render 方法生成新的 VNode，接着调用 patch 方法对比新旧 VNode 来更新 DOM。

Vue.js 2.0 使用了 Virtual DOM 技术，除了在数据变化——更新 DOM 这块有性能方面的提升外，还可以很好地支持服务端渲染技术。接下来就让我们看一下 Vue.js 2.0 的另一大亮点——服务端渲染技术。这里我们只分析实现原理，具体使用需要等到 Vue.js 2.0 正式发布后。

18.3　服务端渲染技术

Vue.js 1.x 版本提倡组件化开发模式，把页面拆分成一个个 Vue 组件，页面渲染工作交给前端 JS 完成。Vue.js 2.0 提供了服务端渲染技术，服务端渲染比起客户端渲染页面，有以下几点优势：

1. 首屏渲染速度更快

客户端渲染的一个缺点是，用户第一次访问页面，此时浏览器没有缓存，需要先从服务端下载 JS。然后再通过 JS 操作动态添加 DOM 并渲染页面，时间较长；而服务端渲染则是，用户第一次访问浏览器可以直接解析 HTML 文档并渲染页面，首屏渲染速度要比客户端渲染快。

2. SEO

服务端渲染可以让搜索引擎更容易读取页面的 meta 信息，以及其他 SEO 相关信息，大大增加了网站在搜索引擎中的可见度。

3. 减少 HTTP 请求

服务端渲染可以把一些动态数据在首次渲染时同步输出到页面；而客户端渲染需要通过 AJAX 等手段异步获取这些数据，这样就相当于多了一次 HTTP 请求。

Vue.js 2.0 提供了两种服务端渲染方式，即普通服务端渲染和流式服务端渲染。接下来让我们看一下这两种服务端渲染的原理。

18.3.1　普通服务端渲染

Vue.js 2.0 提供了 renderToString 接口，可以在服务端把 Vue 组件渲染成模板字符串。我们先看一下 renderString 的用法，源码定义如下：

```
<!--源码目录：benchmarks/ssr/renderToString.js-->
const Vue = require('../../dist/vue.common.js')
const createRenderer = require('../../packages/vue-server-renderer')
const renderToString = createRenderer().renderToString
const gridComponent = require('./common.js')
```

```
console.log('--- renderToString --- ')
const self = (global || root)
self.s = self.performance.now()
renderToString(new Vue(gridComponent), () => {
  console.log('Complete time: ' + (self.performance.now() - self.s).toFixed(2) + 'ms')
  console.log()
})
```

这段代码是运行在 Node.js 环境中的，主要依赖 vue.common.js、vue-server-render。其中 vue.common.js 是 Vue 运行时代码，不包括编译部分；vue-server-render 对外提供 createRenderer 方法，renderToString 是 createRenderer 方法返回值的一个属性，它支持传入 Vue 实例和渲染完成后的回调函数。这里要注意，由于引用的是只包含运行时的 Vue 代码，不包括编译部分，所以 Vue 实例必须包含明确的 render 方法。渲染完成后的回调函数支持两个参数，即 err 和 result，其中 err 表示是否出错；result 表示 DOM 字符串。在实际应用中，我们可以将从回调函数拿到的 result 拼接到模板中。接下来我们看一下 renderToString 的实现，源码定义如下：

```
<!--源码目录：src/server/create-render.js-->
const render = createRenderFunction(modules, directives, isUnaryTag)
return {
    renderToString (
      component: Component,
      done: (err: ?Error, res: ?string) => any
    ): void {
      let result = ''
      let stackDepth = 0
      const write = (str: string, next: Function) => {
        result += str
        if (stackDepth >= MAX_STACK_DEPTH) {
          process.nextTick(() => {
            try { next() } catch (e) {
              done(e)
            }
          })
        } else {
          stackDepth++
          next()
          stackDepth--
        }
      }
      try {
```

```
        render(component, write, () => {
          done(null, result)
        })
      } catch (e) {
        done(e)
      }
    },
  …
  }
```

renderToString 方法支持传入 Vue 实例 component 和渲染完成后的回调函数 done。它定义了 result 变量，同时定义了 write 方法，最后执行 render 方法。render 方法的功能是把 component 转换成模板字符串 str，写入 write 方法中。write 方法不断拼接模板字符串，用 result 变量做存储，然后调用 next 方法。当 component 通过 render 渲染模板字符串完毕后，执行 done，传入 result。整个过程比较核心的步骤就是 render 方法，我们看一下它的实现，源码定义如下：

```
<!--源码目录：src/server/render.js-->
return function render (
    component: Component,
    write: (text: string, next: Function) => void,
    done: Function
  ) {
    renderNode(component._render(), write, done, true)
  }
```

render 方法实际上是执行了 renderNode 方法，并把 component._render()方法生成的 VNode 对象作为参数传入。renderNode 方法定义如下：

```
<!--源码目录：src/server/render.js-->
function renderNode (
    node: VNode,
    write: Function,
    next: Function,
    isRoot: boolean
  ) {
    if (node.componentOptions) {
      const child =
        getCachedComponent(node) ||
        createComponentInstanceForVnode(node)._render()
      child.parent = node
      renderNode(child, write, next, isRoot)
    } else {
```

```
    if (node.tag) {
      renderElement(node, write, next, isRoot)
    } else {
      write(node.raw ? node.text : encodeHTMLCached(node.text), next)
    }
  }
}
```

renderNode 方法首先判断 node 类型。如果是一个 component VNode，则根据这个 node 创建一个组件的实例并调用_render 方法作为当前 node 的 child VNode，然后递归调用 renderNode 方法；如果是一个普通的 DOM VNode 对象，则调用 renderElement 渲染元素；否则就是一个文本节点，直接调用 write 方法。这里使用了 getCachedComponent 方法尝试从缓存中拿 VNode 实例，对于具有相同 cid 的 component VNode，我们只创建一次。真正把 VNode 对象渲染成 DOM 对象的方法是 renderElement，源码定义如下：

```
<!--源码目录：src/server/render.js-->
function renderElement (
   el: VNode,
   write: Function,
   next: Function,
   isRoot: boolean
 ) {
   if (isRoot) {
     if (!el.data) el.data = {}
     if (!el.data.attrs) el.data.attrs = {}
     el.data.attrs['server-rendered'] = 'true'
   }
   const startTag = renderStartingTag(el)
   const endTag = `</${el.tag}>`
   if (isUnaryTag(el.tag)) {
     write(startTag, next)
   } else if (!el.children || !el.children.length) {
     write(startTag + endTag, next)
   } else {
     const children: Array<VNode> = el.children || []
     write(startTag, () => {
       const total = children.length
       let rendered = 0
       function renderChild (child: VNode) {
         renderNode(child, write, () => {
```

```
          rendered++
          if (rendered < total) {
            renderChild(children[rendered])
          } else {
            write(endTag, next)
          }
        }, false)
      }
      renderChild(children[0])
    })
  }
}
```

renderElement 方法的主要功能是把 VNode 对象渲染成 DOM 元素，我们先看一下它的流程图，如图 18-12 所示。

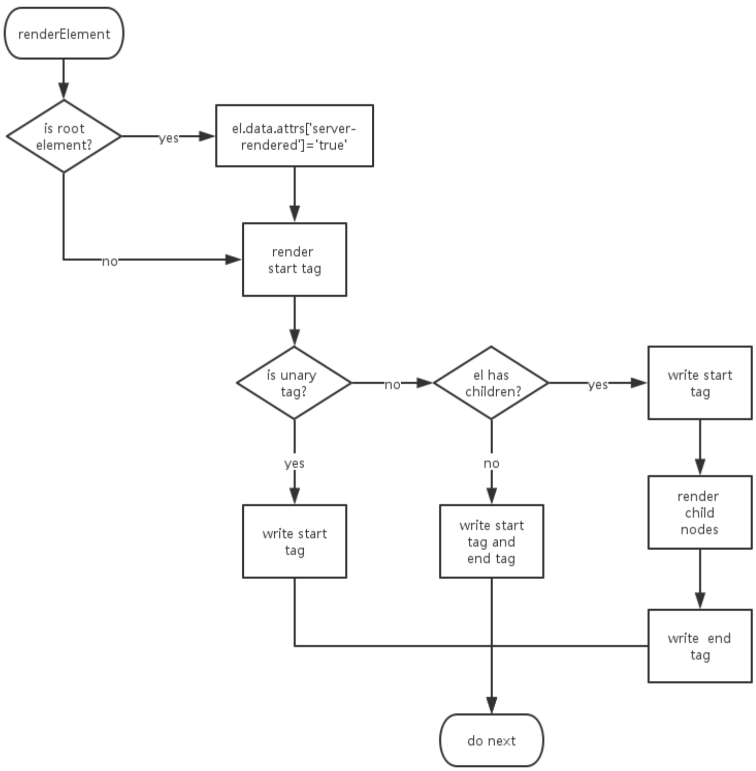


图18-12　renderElement渲染DOM元素流程图

renderElement 首先判断元素是否是根元素，如果是则给元素添加 server-rendered 属性。接

下来渲染开始标签，并对开始标签类型进行判断，如果是自闭合标签如<img>、<input>这样的，则直接通过 write 方法写入开始标签，再执行 next 方法，DOM 渲染完毕。如果元素没有子元素且又不是自闭合标签，则直接通过 write 方法写入开始标签和闭合标签，再执行 next 方法，DOM 渲染完毕；否则就通过 write 方法写入开始标签，接着调用 renderNode 方法渲染所有子节点，再通过 write 方法写入闭合标签，最后执行 next 方法，DOM 渲染完毕。

18.3.2　流式服务端渲染

流式服务端渲染算是 Vue.js 2.0 在服务端渲染上的一大亮点。普通服务端渲染有一个痛点——由于渲染是同步过程，所以如果这个 App 很复杂的话，它可能会阻塞服务器的 event loop，同步服务器渲染在优化不当时甚至会给客户端获得内容的速度带来负面影响。Vue.js 2.0 提供了 renderToStream 接口，在渲染组件时返回一个可读的 stream，可以直接 pipe 到 HTTP Response 中。流式渲染能够确保服务端响应度，也能让用户更快地获得渲染内容。我们先看一下 renderToStream 的用法，源码定义如下：

```
<!--源码目录：benchmarks/ssr/renderToStream.js-->
const Vue = require('../../dist/vue.common.js')
const createRenderer = require('../../packages/vue-server-renderer')
const renderToStream = createRenderer().renderToStream
const gridComponent = require('./common.js')
console.log('--- renderToStream --- ')
const self = (global || root)
self.s = self.performance.now()
const stream = renderToStream(new Vue(gridComponent))
let str = ''
const stats = []
stream.on('data', chunk => {
  str += chunk
  stats.push(self.performance.now())
})
stream.on('end', () => {
  stats.push(self.performance.now())
  stats.forEach((val, index) => {
    const type = index !== stats.length - 1 ? 'Chunk' : 'Complete'
    console.log(type + ' time: ' + (val - self.s).toFixed(2) + 'ms')
  })
  console.log()
})
```

这段代码也是同样运行在 Node.js 环境中的，与 renderToString 不同，renderToStream 会把 Vue 实例渲染成一个可读的 stream。源码演示的是监听数据的读取，并记录读取数据的时间。而在实际应用中，我们可以这样写，代码示例如下：

```
const Vue = require('../../dist/vue.common.js')
const createRenderer = require('../../packages/vue-server-renderer')
const renderToStream = createRenderer().renderToStream
const gridComponent = require('./common.js')
const stream = renderToStream(new Vue(gridComponent))
app.use(function(req, res){
  stream.pipe(res)
})
```

如果代码运行在 Express 框架中，则可以通过 app.use 方法创建 middleware，然后直接把 stream pipe 到 res 中，这样客户端就能很快地获得渲染内容了。接下来我们看一下 renderToStream 的实现，源码定义如下：

```
<!--源码目录：src/server/create-render.js-->
const render = createRenderFunction(modules, directives, isUnaryTag)
return {
    …
    renderToStream (component: Component): RenderStream {
      return new RenderStream((write, done) => {
        render(component, write, done)
      })
    }
  }
```

renderToStream 传入一个 Vue 对象实例，返回的是一个 RenderStream 对象的实例。我们来看一下 RenderStream 对象的实现，源码定义如下：

```
<!--源码目录：src/server/render-stream.js-->
import stream from 'stream'
import { MAX_STACK_DEPTH } from './create-renderer'
/**
 * Original RenderStream implmentation by Sasha Aickin (@aickin)
 * Licensed under the Apache License, Version 2.0
 * Modified by Evan You (@yyx990803)
 */
export default class RenderStream extends stream.Readable {
```

```
buffer: string;
render: Function;
expectedSize: number;
stackDepth: number;
write: Function;
next: Function;
end: Function;
done: boolean;
constructor (render: Function) {
  super()
  this.buffer = ''
  this.render = render
  this.expectedSize = 0
  this.stackDepth = 0
  this.write = (text: string, next: Function) => {
    const n = this.expectedSize
    this.buffer += text
    if (this.buffer.length >= n) {
      this.next = next
      this.pushBySize(n)
    } else {
      // continue rendering until we have enough text to call this.push().
      // sometimes do this as process.nextTick to get out of stack overflows.
      if (this.stackDepth >= MAX_STACK_DEPTH) {
        process.nextTick(() => {
          try { next() } catch (e) {
            this.emit('error', e)
          }
        })
      } else {
        this.stackDepth++
        next()
        this.stackDepth--
      }
    }
  }
  this.end = () => {
    // the rendering is finished; we should push out the last of the buffer.
```

```
      this.done = true
      this.push(this.buffer)
    }
  }
  pushBySize (n: number) {
    const bufferToPush = this.buffer.substring(0, n)
    this.buffer = this.buffer.substring(n)
    this.push(bufferToPush)
  }
  tryRender () {
    try {
      this.render(this.write, this.end)
    } catch (e) {
      this.emit('error', e)
    }
  }
  tryNext () {
    try {
      this.next()
    } catch (e) {
      this.emit('error', e)
    }
  }
  _read (n: number) {
    this.expectedSize = n
    // it's possible that the last chunk added bumped the buffer up to > 2 * n,
    // which means we will need to go through multiple read calls to drain it
    // down to < n.
    if (this.done) {
      this.push(null)
      return
    }
    if (this.buffer.length >= n) {
      this.pushBySize(n)
      return
    }
    if (!this.next) {
      // start the rendering chain.
```

```
      this.tryRender()
    } else {
      // continue with the rendering.
      this.tryNext()
    }
  }
}
```

RenderStream 继承了 node 中的可读流 stream.Readable，它是一个自定义的可读流。所有可读流的实现都必须提供一个_read 方法从底层资源抓取数据。注意，这个方法是以下画线开头的，表明它是一个内部方法，并且不应该被用户程序直接调用。可读流通过 push 方法向队列中插入一些数据，而且至少需要调用一次 push(chunk)方法，_read 方法才会再次被调用。当 push 方法传入 null 参数时，它会触发数据结束信号（EOF）。接下来我们详细介绍一下 RenderStream 类的实现。

1. Constructor(render)

构造函数方法首先通过 super 方法调用了父类的构造函数，以便缓冲设定能被正确地初始化，然后初始化 buffer、expectedSize、stackDepth、render 等字段。其中 buffer 表示缓冲区字符串；expectedSize 为每次期望往读取队列中插入内容的大小；stackDepth 记录 write 方法递归调用的深度；render 保存传入的 render 方法。最后分别定义了 write 和 end 方法。

2. Write(text, next)

write 方法首先把 text 拼接到 buffer 缓冲区，然后判断 buffer.length。如果 buffer.length 大于 expectedSize，则用 this.next 保存 next 方法，同时调用 this.pushBySize(n)把缓冲区内容推入读取队列中；如果 buffer.length 小于 expectedSize，则调用 next 方法继续渲染组件，在调用 next 之前会判断 stackDepth 的大小；如果 stackDepth 大于或等于 MAX_STACK_DEPTH，则说明 write 方法递归调用的深度已经超过设定值，通过 process.nextTick 的方式调用 next 方法，避免堆栈溢出。

3. end

end 方法设置 this.done 标志位为 true，标识组件的渲染已经完毕。接着调用 this.push(buffer)方法把缓冲区剩余内容推入读取队列中。

4. _read(n)

我们先来看一下它的实现逻辑，如图 18-13 所示。

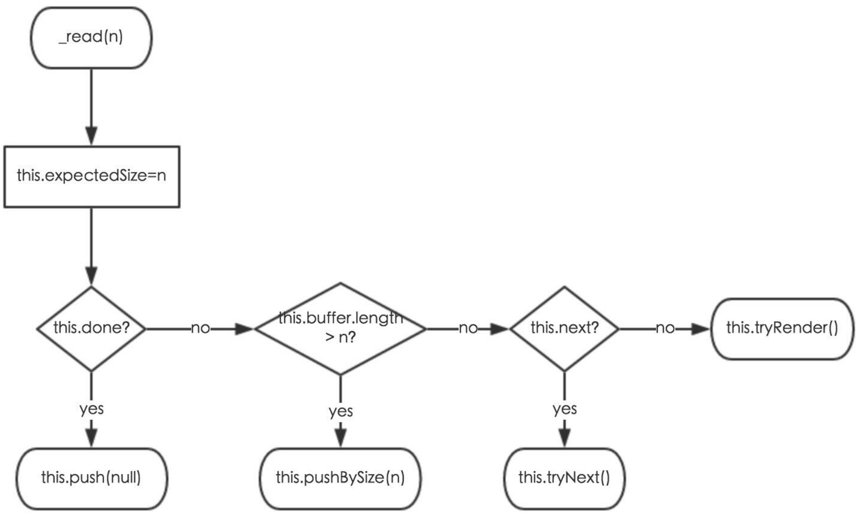


图18-13　_read方法实现逻辑图

_read(n)方法首先通过 this.expectedSize=n 记录了本次读取的字节数，然后判断 this.done。如果为 true 则表示组件已渲染完毕，这时候调用 this.push(null)触发数据结束信号（EOF）。如果渲染未完成，接下来判断 this.buffer.length。如果大于 n 则说明缓冲区的字符串长度足够，可以调用 this.pushBySize(n)方法把缓冲区内容推入读取队列中；如果 this.buffer.length 小于 n，接下来判断 this.next。如果为 false 则表示开始渲染组件，调用 this.tryRender()开始渲染组件；否则调用 this.tryNext()继续渲染组件。

5. PushBySize(n)

pushBySize 方法截取 buffer 缓冲区前 *n* 个长度的数据，推入到读取队列中，同时更新 buffer 缓冲区，删除前 *n* 条数据。

6. tryRender

tryRender 方法调用 this.render(this.write, this.end)方法开始渲染组件，这个 render 方法是在初始化 ReadStream 方法时传入的，源码定义如下：

```
renderToStream (component: Component): RenderStream {
   return new RenderStream((write, done) => {
     render(component, write, done)
   })
}
```

这个方法中调用的 render(component,write,done)和 renderToString 使用的 render 方法是同一

个,只不过这里的 write 和 done 对应的是 RenderToStream 类里定义的 this.write 和 this.end 方法。

7. tryNext

tryNext 方法调用 this.next()方法继续渲染组件。

最后简单做一下回顾。首先先调用 renderToStream(new Vue(option))创建好 stream 对象后，通过 stream.pipe()方法把数据发送到一个 WritableStream 中 ，会触发 RenderToStream 内部_read 方法的调用，不断把渲染的组件数据推入读取队列中，这个 WritableStream 就可以不断地读取到组件的数据，然后输出。这样就实现了流式服务端渲染技术。

18.4　总结

本章首先介绍了 Vue.js 2.0 相对于 1.x 版本的 API 变更，让我们了解到 Vue.js 2.0 在 API 设计上更加简洁，新手上手成本更低，更易于掌握框架。接着介绍了 Vue.js 2.0 的两大新特性——Virtual DOM 和服务端渲染。了解这些技术的实现原理和细节，会对我们今后使用 Vue.js 2.0 进行开发，遇到一些性能优化的场景时有一定的帮助。

第 19 章

源码篇——util

Vue.js 内部封装了 util，提供了一些常用的工具方法。了解本章内容，可以避免开发者再额外引用第三方框架增加代码量。util 一共分成 6 部分：env、dom、components、lang、debug 和 options，整体结构如图 19-1 所示。

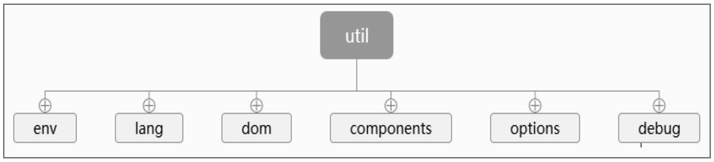


图19-1　util整体结构

19.1　env

env 如图 19-2 所示。

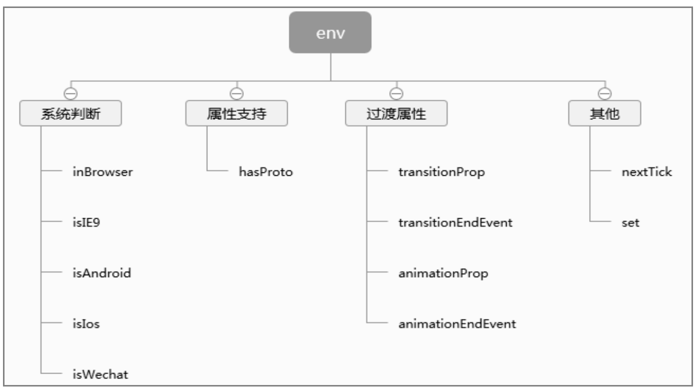


图19-2　env

19.1.1 系统判断

1. inBrowser

判断是否为浏览器环境，源码定义如下：

```
<!--源码目录：src/util/env.js 7 行-->
export const inBrowser =
  typeof window !== 'undefined' &&
  Object.prototype.toString.call(window) !== '[object Object]'
```

2. isIE9

判断是否为 IE 9，源码定义如下：

```
<!--源码目录：src/util/env.js 16 行-->
const UA = inBrowser && window.navigator.userAgent.toLowerCase()
export const isIE9 = UA && UA.indexOf('msie 9.0') > 0
```

3. isAndroid

判断是否为安卓，源码定义如下：

```
<!--源码目录：src/util/env.js 17 行-->
export const isAndroid = UA && UA.indexOf('android') > 0
```

4. isIos

判断是否为 iOS，源码定义如下：

```
<!--源码目录：src/util/env.js 18 行-->
export const isIos = UA && /(iphone|ipad|ipod|ios)/i.test(UA)
```

5. isWechat

判断是否为微信，源码定义如下：

```
<!--源码目录：src/util/env.js 19 行-->
export const isWechat = UA && UA.indexOf('micromessenger') > 0
```

19.1.2 属性支持

hasProto——是否支持__proto__属性，源码定义如下：

```
<!--源码目录：src/util/env.js 7 行-->
export const hasProto = '__proto__' in {}
```

19.1.3　过渡属性

1. transitionProp

```
<!--源码目录：src/util/env.js 34 行-->
const isWebkitTrans =
  window.ontransitionend === undefined &&
  window.onwebkittransitionend !== undefined
const isWebkitAnim =
  window.onanimationend === undefined &&
  window.onwebkitanimationend !== undefined
transitionProp = isWebkitTrans
  ? 'WebkitTransition': 'transition'
```

2. transitionEndEvent

transitionend 事件在 CSS 过渡完成后触发。

注：如果过渡在完成前被移除，例如 CSS transition-property 属性被移除，过渡事件将不被触发。

```
<!--源码目录：src/util/env.js 37 行-->
transitionEndEvent = isWebkitTrans? 'webkitTransitionEnd': 'transitionend'
```

3. animationProp

CSS 动画效果将会影响元素相对应的 css 值，在整个动画过程中，元素的变化属性值完全是由 animation 控制的，动画后面的属性值会覆盖前面的属性值。animation 类似于 transition 属性，它们都是随着时间改变元素的属性值。其主要区别是 transition 需要触发一个事件（hover 或 click 事件等）才会随时间改变其 css 属性值；而 animation 在不需要触发任何事件的情况下也可以显式地随着时间变化来改变元素的 css 属性值，从而达到一种动画效果。

```
<!--源码目录：src/util/env.js 40 行-->
animationProp = isWebkitAnim? 'WebkitAnimation': 'animation'
```

4. animationEndEvent

animationend 事件在 CSS 动画完成后触发。

```
<!--源码目录：src/util/env.js 43 行-->
animationEndEvent = isWebkitAnim? 'webkitAnimationEnd' : 'animationend'
```

19.1.4　nextTick

异步执行，在 Vue.js 内部，Vue.js 会使用 MutationObserver 来实现队列的异步处理，如果不支持则会回退到 setTimeout(fn, 0) 。当 Vue.nextTick 的回调函数执行时，DOM 已经是更新后的状态了。

- cb {Function}
- ctx {Object}

```
<!--源码目录：src/util/env.js 65 行-->
(function () {
  var callbacks = []
  var pending = false
  var timerFunc
  function nextTickHandler () {
    pending = false
    var copies = callbacks.slice(0)
    callbacks = []
    for (var i = 0; i < copies.length; i++) {
      copies[i]()
    }
  }
  if (typeof MutationObserver !== 'undefined' && !(isWechat && isIos)) {
    var counter = 1
    var observer = new MutationObserver(nextTickHandler)
    var textNode = document.createTextNode(counter)
    observer.observe(textNode, {
      characterData: true
    })
    timerFunc = function () {
      counter = (counter + 1) % 2
      textNode.data = counter
    }
  } else {
    // webpack attempts to inject a shim for setImmediate
    // if it is used as a global, so we have to work around that to
```

```
    // avoid bundling unnecessary code.
    const context = inBrowser
      ? window
      : typeof global !== 'undefined' ? global : {}
    timerFunc = context.setImmediate || setTimeout
  }
  return function (cb, ctx) {
    var func = ctx
      ? function () { cb.call(ctx) }
      : cb
    callbacks.push(func)
    if (pending) return
    pending = true
    timerFunc(nextTickHandler, 0)
  }
})()
```

Vue 实例的$nextTick 方法其实调用的就是此方法，源码定义如下：

```
<!--源码目录：src/instance/api/dom.js 25 行-->
Vue.prototype.$nextTick = function (fn) {
  nextTick(fn, this)
}
```

Vue.js 的过渡动画队列同样使用此方法，源码定义如下：

```
<!--源码目录：src/transition/queue.js 12 行-->
function pushJob (job) {
  queue.push(job)
  if (!queued) {
    queued = true
    nextTick(flush) // flush 方法为执行队列所有方法，在过渡之前
  }
}
```

Vue.js 的 v-text 指令中也使用了此方法，源码定义如下：

```
<!--源码目录：src/derictives/public/model/text.js 111 行-->
if (!lazy && isIE9) {
  this.on('cut', function () {
    nextTick(self.listener)
```

```
  })
  this.on('keyup', function (e) {
    if (e.keyCode === 46 || e.keyCode === 8) {
      self.listener()
    }
  })
 }
```

19.1.5　set

创建 set 简单对象，挂载属性 set、add、clear、has 方法。目前暂时在 watcher.js 中使用，源码定义如下：

```
<!--源码目录：src/util/env.js 131 行-->
if (typeof Set !== 'undefined' && Set.toString().match(/native code/)) {
  // use native Set when available.
  _Set = Set
} else {
  // a non-standard Set polyfill that only works with primitive keys.
  _Set = function () {
    this.set = Object.create(null)
  }
  _Set.prototype.has = function (key) {
    return this.set[key] !== undefined
  }
  _Set.prototype.add = function (key) {
    this.set[key] = 1
  }
  _Set.prototype.clear = function () {
    this.set = Object.create(null)
  }
}
```

19.2　dom

在本节中，我们将分 5 部分进行详细的讲解，如图 19-3 所示。

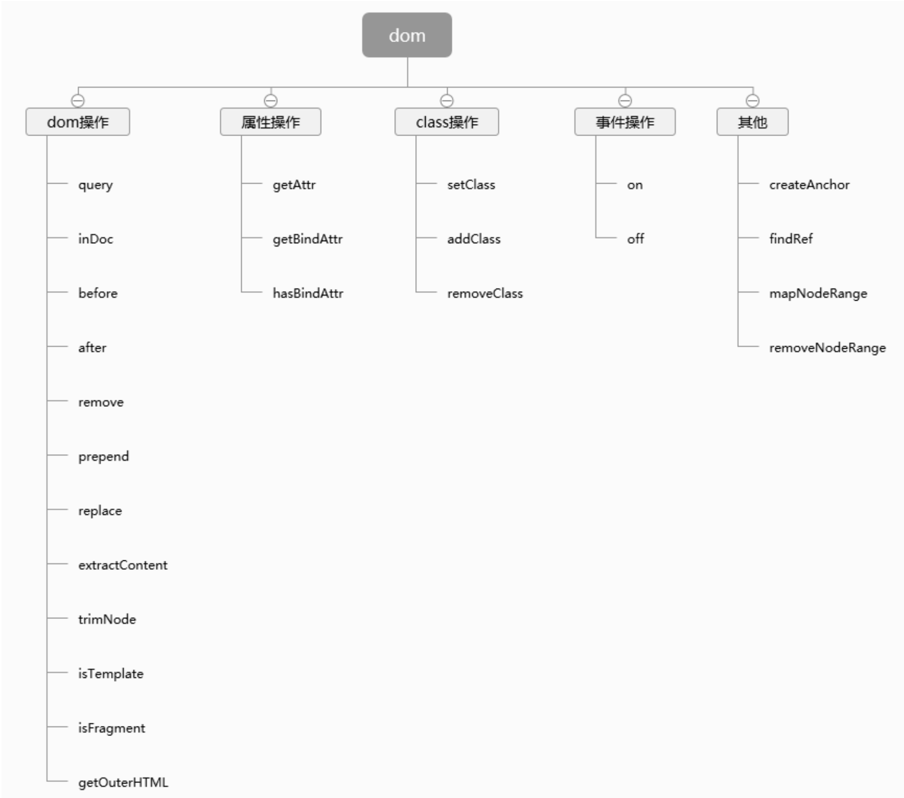


图19-3　dom

19.2.1　dom 操作

1. query

查找 dom 元素，使用 document.querySelector 方法返回文档中匹配指定 CSS 选择器的元素集合中的第一个元素（兼容 IE 8 及以上版本）。

- el {String|Element}

```
<!--源码目录：src/util/dom.js 14 行-->
function query (el) {
  if (typeof el === 'string') {
    var selector = el
    el = document.querySelector(el)
    if (!el) {
      process.env.NODE_ENV !== 'production' && warn(
        'Cannot find element: ' + selector
```

```
      )
    }
  }
  return el
}
```

2. inDoc

是否在文档中，运用了 ownerDocument.documentElement 属性，其中 ownerDocument 是文档，documentElement 是根节点，ownerDocument 下包含两个节点，即 doctype 和 documentElement。

- node {node}

```
<!--源码目录：src/util/dom.js 39 行-->
function inDoc (node) {
  if (!node) return false
   var doc = node.ownerDocument.documentElement
   var parent = node.parentNode
   return doc === node ||
     doc === parent ||
     !!(parent && parent.nodeType === 1 && (doc.contains(parent)))
}
```

3. before

在 target 节点前插入 el 元素。

- el {Element}
- target {Element}

```
<!--源码目录：src/util/dom.js 100 行-->
function before (el, target) {
  target.parentNode.insertBefore(el, target)
}
```

4. after

在 target 节点后插入 el 元素。

- el {Element}
- target {Element}

```
<!--源码目录：src/util/dom.js 111 行-->
function after (el, target) {
  if (target.nextSibling) {
```

```
    before(el, target.nextSibling)
  } else {
    target.parentNode.appendChild(el)
  }
}
```

5. prepend

在 target 节点最前面插入 el 元素。

- el {Element}
- target {Element}

```
<!--源码目录：src/util/dom.js 136 行-->
function prepend (el, target) {
  if (target.firstChild) {
    before(el, target.firstChild)
  } else {
    target.appendChild(el)
  }
}
```

6. extractContent

将元素内容提取到一个 div 元素或文本碎片中（取决于 asFragment 参数）。

- el {Element}
- asFragment {Boolean}

```
<!--源码目录：src/util/dom.js 268 行-->
function extractContent (el, asFragment) {
  var child
  var rawContent
  if (isTemplate(el) && isFragment(el.content)) {
    el = el.content
  }
  if (el.hasChildNodes()) {
    trimNode(el)
    rawContent = asFragment
      ? document.createDocumentFragment()
      : document.createElement('div')
    while (child = el.firstChild) {
      rawContent.appendChild(child)
```

```
    }
  }
  return rawContent
}
```

Vue.js 在组件中内联模板参数使用的该方法，将组件中包含的元素作为内联模板，源码定义如下：

```
<!--源码目录：src/directive/internal/component.js 41 行-->
bind () {
  if (!this.el.__vue__) {
    // keep-alive cache
    this.keepAlive = this.params.keepAlive
    if (this.keepAlive) {
      this.cache = {}
    }
    // check inline-template
    if (this.params.inlineTemplate) {
      // extract inline template as a DocumentFragment
      this.inlineTemplate = extractContent(this.el, true)
    }
    ……
},
```

用 inlineTemplate 赋值给组件的模板 template 属性，源码定义如下：

```
<!--源码目录：src/directive/internal/component.js 201 -->
build (extraOptions) {
  var cached = this.getCached()
  if (cached) {
    return cached
  }
  if (this.Component) {
    // default options
    var options = {
      name: this.ComponentName,
      el: cloneNode(this.el),
      template: this.inlineTemplate
      ……
    }
    if (extraOptions) {
      extend(options, extraOptions)
    }
    ……
```

```
    return child
  }
},
```

7. remove

删除 el 元素。

- el {Element}

```
<!--源码目录：src/util/dom.js 125 行-->
function remove (el) {
  el.parentNode.removeChild(el)
}
```

8. replace

el 元素替换 target。

- target {Element}
- el {Element}

```
<!--源码目录：src/util/dom.js 151 行-->
function replace (target, el) {
  var parent = target.parentNode
  if (parent) {
    parent.replaceChild(el, target)
  }
}
```

9. trimNode

清除 node 节点内首尾空文本或注释节点，使用 isTrimmable 判断是否为空文本或注释节点，该方法在模板解析中使用最多。

- node {Node}

```
<!--源码目录：src/util/dom.js 296 行-->
function trimNode (node) {
  var child;
  while (child = node.firstChild, isTrimmable(child)) {
    node.removeChild(child)
  }
```

```
  while (child = node.lastChild, isTrimmable(child)) {
    node.removeChild(child)
  }
}
function isTrimmable (node) {
  return node && (
    (node.nodeType === 3 && !node.data.trim()) ||
    node.nodeType === 8
  )
}
```

10. isTemplate

是否为 template 模板元素。

- el {Element}

```
<!--源码目录：src/util/dom.js 323行-->
function isTemplate (el) {
  return el.tagName &&
    el.tagName.toLowerCase() === 'template'
}
```

11. isFragment

是否为轻量级的 Document 对象，能够容纳文档的某个部分。

- node {Node}

```
<!--源码目录：src/util/dom.js 431行-->
function isFragment (node) {
  return node && node.nodeType === 11
}
```

12. getOuterHTML

获取元素 outerHTML，如果不支持则获取 innerHTML，用 div 元素包裹。

- el {Element}

```
<!--源码目录：src/util/dom.js 443行-->
function getOuterHTML (el) {
  if (el.outerHTML) {
    return el.outerHTML
  } else {
```

```
    var container = document.createElement('div')
    container.appendChild(el.cloneNode(true))
    return container.innerHTML
  }
}
```

Vue.js 只用在 FragmentFactory 中：

```
<!--源码目录：src/fragment/factory.js 17 行-->
function FragmentFactory (vm, el) {
  this.vm = vm
  var template
  var isString = typeof el === 'string'
  if (isString || isTemplate(el) && !el.hasAttribute('v-if')) {
    template = parseTemplate(el, true)
  } else {
    template = document.createDocumentFragment()
    template.appendChild(el)
  }
  this.template = template
  // linker can be cached, but only for components
  var linker
  var cid = vm.constructor.cid
  if (cid > 0) {
    var cacheId = cid + (isString ? el : getOuterHTML(el))
    linker = linkerCache.get(cacheId)
    if (!linker) {
      linker = compile(template, vm.$options, true)
      linkerCache.put(cacheId, linker)
    }
  } else {
    linker = compile(template, vm.$options, true)
  }
  this.linker = linker
}
```

19.2.2　属性操作

1. getAttr

在 node 元素上获取并移除_attr 属性。

- node {Node}
- _arrt {String}

```
<!--源码目录：src/util/dom.js 55 行-->
function getAttr (node, _attr) {
  var val = node.getAttribute(_attr)
  if (val !== null) {
    node.removeAttribute(_attr)
  }
  return val
}
```

2. getBindAttr

获取：name 或 v-bind:name 的属性值。

- node {Node}
- name {String}

```
<!--源码目录：src/util/dom.js 71 行-->
function getBindAttr (node, name) {
  var val = getAttr(node, ':' + name)
  if (val === null) {
    val = getAttr(node, 'v-bind:' + name)
  }
  return val
}
```

此方法在以下情况中使用：①编译 props 属性；②指令中_setupParams 方法；③获取动态组件 is 属性值。我们看一下在指令中的使用方法，源码定义如下：

```
<!--源码目录：src/directive.js 193 行-->
Directive.prototype._setupParams = function () {
  if (!this.params) {
    return
  }
  var params = this.params
  this.params = Object.create(null)
  var i = params.length
  var key, val, mappedKey
  while (i--) {
```

```
    key = hyphenate(params[i])
    mappedKey = camelize(key)
    val = getBindAttr(this.el, key)
    if (val != null) {
      // 为动态参数绑定$watch 方法
      this._setupParamWatcher(mappedKey, val)
    } else {
      // 静态
      val = getAttr(this.el, key)
      if (val != null) {
        this.params[mappedKey] = val === '' ? true : val
      }
    }
  }
}
```

3. hasBindAttr

node 元素上是否有绑定的 name 属性。

- node {Node}
- name {String}

```
<!--源码目录：src/util/dom.js 87 行-->
function hasBindAttr (node, name) {
  return node.hasAttribute(name) ||
    node.hasAttribute(':' + name) ||
    node.hasAttribute('v-bind:' + name)
}
```

19.2.3　class 操作

1. setClass

设置 class 名称（在 IE 9 中，如果 el 元素有:class 属性，将忽略该 class 属性；然而在 PhantomJS，设置 className 无效）。

- el {Element}
- cls {String}

```
<!--源码目录：src/util/dom.js 209行-->
function setClass (el, cls) {
  if (isIE9 && !/svg$/.test(el.namespaceURI)) {
    el.className = cls
  } else {
    el.setAttribute('class', cls)
  }
}
```

2. addClass

添加 className，其中 classList 属性返回元素的类名，作为 DOMTokenList 对象，该属性用于在元素中添加、移除及切换 CSS 类。classList 属性是只读的，但我们可以使用 add()和 remove()方法修改它（IE 10 及以上版本）。

- el {Element}
- cls {String}

```
<!--源码目录：src/util/dom.js 225行-->
function addClass (el, cls) {
  if (el.classList) {
    el.classList.add(cls)
  } else {
    var cur = ' ' + getClass(el) + ' '
    if (cur.indexOf(' ' + cls + ' ') < 0) {
      setClass(el, (cur + cls).trim())
    }
  }
}
```

3. removeClass

删除 el 元素的 className，如果没有 className，则移除元素的 class 属性。

- el {Element}
- cls {String}

```
<!--源码目录：src/util/dom.js 243行-->
function removeClass (el, cls) {
  if (el.classList) {
    el.classList.remove(cls)
```

```
  } else {
    var cur = ' ' + getClass(el) + ' '
    var tar = ' ' + cls + ' '
    while (cur.indexOf(tar) >= 0) {
      cur = cur.replace(tar, ' ')
    }
    setClass(el, cur.trim())
  }
  if (!el.className) {
    el.removeAttribute('class')
  }
}
```

19.2.4　事件操作

1. on

el 绑定 event 事件监听（useCapture 指定事件是否在捕获或冒泡阶段执行，true 表示事件句柄在捕获阶段执行，默认值为 false）（IE 9 及以上版本）

- el {Element}
- event {String}
- cb {Function}
- useCapture {Boolean}

```
<!--源码目录：src/util/dom.js 167 行-->
function on (el, event, cb, useCapture) {
  el.addEventListener(event, cb, useCapture)
}
```

在指令中使用 on 绑定事件的代码示例如下：

```
<!--源码目录：src/directive.js 296 行-->
Directive.prototype.on = function (event, handler, useCapture) {
  on(this.el, event, handler, useCapture)
  ;(this._listeners || (this._listeners = []))
    .push([event, handler])
}
```

2. off

移除事件监听。

- el {Element}
- event {String}
- cb {Function}

```
<!--源码目录：src/util/dom.js 179行-->
function off (el, event, cb) {
  el.removeEventListener(event, cb)
}
```

19.2.5 其他

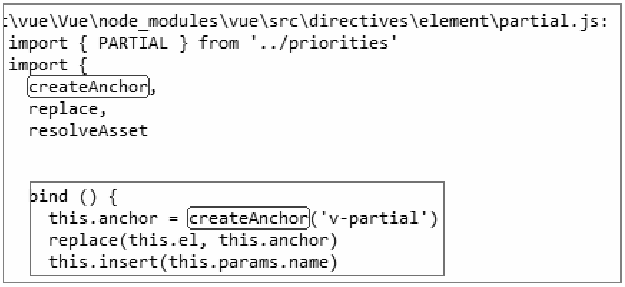
1. createAnchor

创建一个“执行插入/删除锚点”，使用场景：①片段实例；②v-html；③v-if；④v-for；⑤组件。

- content { String }
- persist {Boolean}

```
<!--源码目录：src/util/dom.js 346行-->
function createAnchor (content, persist) {
  var anchor = config.debug
    ? document.createComment(content)
    : document.createTextNode(persist ? ' ' : '')
  anchor.__v_anchor = true
  return anchor
}
```

v-partial 的使用如图 19-4 所示，其他的使用类似。



```
:\vue\Vue\node_modules\vue\src\directives\element\partial.js:
import { PARTIAL } from '../priorities'
import {
  createAnchor,
  replace,
  resolveAsset

  bind () {
    this.anchor = createAnchor('v-partial')
    replace(this.el, this.anchor)
    this.insert(this.params.name)
```

图19-4　createAnchor

2. findRef

在 node 元素中找到 v-ref 绑定的值。

- node {Element}

```
<!--源码目录：src/util/dom.js 361 行-->
var refRE = /^v-ref:/
export function findRef (node) {
  if (node.hasAttributes()) {
    var attrs = node.attributes
    for (var i = 0, l = attrs.length; i < l; i++) {
      var name = attrs[i].name
      if (refRE.test(name)) {
        return camelize(name.replace(refRE, ''))
      }
    }
  }
}
```

3. mapNodeRange

在范围兄弟节点内调用 op 函数。

- node {Node}
- end {Node}
- op {Function}

```
<!--源码目录：src/util/dom.js 382 行-->
function mapNodeRange (node, end, op) {
  var next
  while (node !== end) {
```

```
    next = node.nextSibling
    op(node)
    node = next
  }
  op(end)
}
```

4. removeNodeRange

依次移除范围内兄弟元素并移入 frag 元素中，其中用到了过渡中的 removeWithTransitio 方法，整体移除完毕后调用回调函数。

- start {Node}
- end {Node}
- vm {Vue}
- frag {DocumentFragment}
- cb {Function}

```
<!--源码目录：src/util/dom.js 404 行-->
function removeNodeRange (start, end, vm, frag, cb) {
  var done = false
  var removed = 0
  var nodes = []
  // 获取 start 至 end 的所有兄弟元素
  mapNodeRange(start, end, function (node) {
    if (node === end) done = true
    nodes.push(node)
    removeWithTransition(node, vm, onRemoved)
  })
  function onRemoved () {
    removed++
    if (done && removed >= nodes.length) {
      for (var i = 0; i < nodes.length; i++) {
        frag.appendChild(nodes[i])
      }
      cb && cb()
    }
  }
}
```

19.3　lang

lang 如图 19-5 所示。



图19-5　lang

19.3.1　对象操作

1. set

设置对象属性，添加新属性触发更新。

- obj {Object}
- key {String}
- val { * }

```
<!--源码目录：src/util/lang.js 12 行-->
function set (obj, key, val) {
  if (hasOwn(obj, key)) {   // 如果是自有属性，为属性赋值，返回
    obj[key] = val
```

```
    return
  }
  if (obj._isVue) {// 是 vue 实例，递归调用该函数
    set(obj._data, key, val)
    return
  }
  var ob = obj.__ob__
  if (!ob) {// 不是被监测对象，将 val 赋值到 obj[key]
    obj[key] = val
    return
  }
  // 是被监测对象
  ob.convert(key, val)// defineProperty
  ob.dep.notify()
  if (ob.vms) {
    var i = ob.vms.length
    while (i--) {
      var vm = ob.vms[i]
      vm._proxy(key)
      vm._digest()
    }
  }
  return val
}
```

2. del

删除对象属性，触发更新。

- obj {Object}
- key {String}

```
<!--源码目录：src/util/lang.js 46 行-->
function del (obj, key) {
  if (!hasOwn(obj, key)) {
    return
  }
  delete obj[key]
  var ob = obj.__ob__
```

```
  if (!ob) {
    if (obj._isVue) {
      delete obj._data[key]
      obj._digest()
    }
    return
  }
  ob.dep.notify()
  if (ob.vms) {
    var i = ob.vms.length
    while (i--) {
      var vm = ob.vms[i]
      vm._unproxy(key)
      vm._digest()
    }
  }
}
```

3. hasOwn

判断是否为自有属性。

- obj {Object}
- key {String}

```
<!--源码目录：src/util/lang.js 78 行-->
function hasOwn (obj, key) {
  return Object.prototype.hasOwnProperty.call(obj, key)
}
```

4. extend

扩展对象属性。

- to {Object}
- from {Object}

```
<!--源码目录：src/util/lang.js 260 行-->
function extend (to, from) {
  var keys = Object.keys(from)
```

```
  var i = keys.length
  while (i--) {
    to[keys[i]] = from[keys[i]]
  }
  return to
}
```

5. isObject

是否为对象。

- obj { * }

```
<!--源码目录：src/util/lang.js 278 行-->
function isObject (obj) {
  return obj !== null && typeof obj === 'object'
}
```

6. isPlainObject

是否为对象字面量。

- obj { * }

```
<!--源码目录：src/util/lang.js 292 行-->
var toString = Object.prototype.toString
var OBJECT_STRING = '[object Object]'
function isPlainObject (obj) {
  return toString.call(obj) === OBJECT_STRING
}
```

7. def

将属性添加到对象，或修改现有属性的特性。关于 Object.defineProperty 详见 30.3 节。

- obj {Object}
- key {String}
- val { * }
- enumerable {Boolean}

```
<!--源码目录：src/util/lang.js 314 行-->
function def (obj, key, val, enumerable) {
```

```
  Object.defineProperty(obj, key, {
    value: val,
    enumerable: !!enumerable,
    writable: true,
    configurable: true
  })
}}
```

19.3.2　名称转换

1. classify

将-、_、/ 的命名转换成驼峰命名方式。

- str {String}

```
<!--源码目录：src/util/lang.js 212 行-->
var classifyRE = /(?:^|[-_\/])(\w)/g
function classify (str) {
  return str.replace(classifyRE, toUpper)
}
```

2. hyphenate

将驼峰命名方式转换为-。

- str {String}

```
<!--源码目录：src/util/lang.js 193 行-->
var hyphenateRE = /([a-z\d])([A-Z])/g
export function hyphenate (str) {
  return str
    .replace(hyphenateRE, '$1-$2')
    .toLowerCase()
}
```

3. camelize

将-命名方式转换驼峰式。

- str {String}

```
<!--源码目录：src/util/lang.js 176 行-->
var camelizeRE = /-(\w)/g
```

```
export function camelize (str) {
  return str.replace(camelizeRE, toUpper)
}
```

19.3.3 数组操作

indexOf——返回 obj 在 Array 中的索引位置，没有则返回-1。

- arr {Array}
- obj { * }

```
<!--源码目录：src/util/lang.js 363 行-->
function indexOf (arr, obj) {
  var i = arr.length
  while (i--) {
    if (arr[i] === obj) return i
  }
  return -1
}
```

19.3.4 类型转换

1. _toString

字符串转换。

- Value { * }

```
<!--源码目录：src/util/lang.js 114 行-->
function _toString (value) {
  return value == null
    ? ''
    : value.toString()
}
```

2. toNumber

数字转换。

- Value { * }

```
<!--源码目录：src/util/lang.js 128 行-->
```

```
function toNumber (value) {
  if (typeof value !== 'string') {
    return value
  } else {
    var parsed = Number(value)
    return isNaN(parsed)
      ? value
      : parsed
  }
}
```

3. toBoolean

布尔转换。

- Value { * }

```
<!--源码目录：src/util/lang.js 146 行-->
function toBoolean (value) {
  return value === 'true'
    ? true
    : value === 'false'
      ? false
      : value
}
```

4. toArray

数组转换。

- Value { * }

```
<!--源码目录：src/util/lang.js 243 行-->
function toArray (list, start) {
  start = start || 0
  var i = list.length - start
  var ret = new Array(i)
  while (i--) {
    ret[i] = list[i + start]
  }
  return ret
}
```

19.3.5　方法绑定

bind——方法绑定。

- fn { Function }
- ctx {Object}

```
<!--源码目录：src/util/lang.js 224 行-->
function bind (fn, ctx) {
  return function (a) {
    var l = arguments.length
    return l
      ? l > 1
        ? fn.apply(ctx, arguments)
        : fn.call(ctx, a)
      : fn.call(ctx)
  }
}
```

19.3.6　其他

1. debounce

此方法只用于 input 输入后，wait 毫秒后调用 func 方法。

- func {Function}
- wait {Number}

```
<!--源码目录：src/util/lang.js 332 行-->
function debounce (func, wait) {
  var timeout, args, context, timestamp, result
  var later = function () {
    var last = Date.now() - timestamp
    if (last < wait && last >= 0) {
      timeout = setTimeout(later, wait - last)
    } else {
      timeout = null
      result = func.apply(context, args)
      if (!timeout) context = args = null
```

```
    }
  }
  return function () {
    context = this
    args = arguments
    timestamp = Date.now()
    if (!timeout) {
      timeout = setTimeout(later, wait)
    }
    return result
  }
}
```

2．stripQuotes

去除 str 中的前后引号。

- str {String}

```
<!--源码目录：src/util/lang.js 161 行-->
function stripQuotes (str) {
  var a = str.charCodeAt(0)
  var b = str.charCodeAt(str.length - 1)
  return a === b && (a === 0x22 || a === 0x27)
    ? str.slice(1, -1)
    : str
}
```

参考表 19-1。

表 19–1　字符表

ASCII	全角字符	Unicode	半角字符	Unicode
0x20	""空格	U+3000	" "空格	U+0020
0x21	！	U+ff01	!	U+0021
0x22	＂	U+ff02	"	U+0022
0x23	＃	U+ff03	#	U+0023
0x24	＄	U+ff04	$	U+0024
0x25	％	U+ff05	%	U+0025
0x26	＆	U+ff06	&	U+0026
0x27	＇	U+ff07	'	U+0027

续表

ASCII	全角字符	Unicode	半角字符	Unicode
0x28	（	U+ff08	(	U+0028
0x29	）	U+ff09	)	U+0029
0x2a	＊	U+ff0a	*	U+002a
0x2b	＋	U+ff0b	+	U+002b

也可以通过 String.fromCharCode()方法来完成。

3. cancellable

可以撤销的异步回调函数。

- fn {Function}

```
<!--源码目录：src/util/lang.js 378 行-->
function cancellable (fn) {
  var cb = function () {
    if (!cb.cancelled) {
      return fn.apply(this, arguments)
    }
  }
  cb.cancel = function () {
    cb.cancelled = true
  }
  return cb
}
```

4. looseEqual

判断 a 和 b 是否相等（非严格意义上的===）。

- a { * }
- b { * }

```
<!--源码目录：src/util/lang.js 399 行-->
function looseEqual (a, b) {
  /* eslint-disable eqeqeq */
  return a == b || (
    isObject(a) && isObject(b)
      ? JSON.stringify(a) === JSON.stringify(b)
      : false
```

```
  )
  /* eslint-enable eqeqeq */
}
```

5. isLiteral

检查表达式是否为字面量。

- exp {String}

```
<!--源码目录：src/util/lang.js 90 行-->
var literalValueRE = /^\s?(true|false|-?[\d\.]+|'[^']*'|"[^"]*")\s?$/
function isLiteral (exp) {
  return literalValueRE.test(exp)
}
```

6. isReserved

检查 str 是否以$ 或 _ 开头。

- str { String }

```
<!--源码目录：src/util/lang.js 101 行-->
function isReserved (str) {
  var c = (str + '').charCodeAt(0)
  return c === 0x24 || c === 0x5F
}
```

19.4　components

components 如图 19-6 所示。

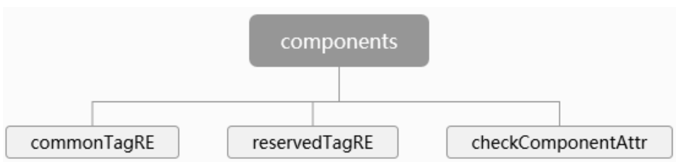


图19-6　components

1. commonTagRE

是否为普通元素。

```
<!--源码目录：src/util/components.js 5 行-->
commonTagRE = /^(div|p|span|img|a|b|i|br|ul|ol|li|h1|h2|h3|h4|h5|h6|code|pre|table|th|
td|tr|form|label|input|select|option|nav|article|section|header|footer)$/i
```

2. reservedTagRE

是否为自定义元素。

```
<!--源码目录：src/util/components.js 6 行-->
reservedTagRE = /^(slot|partial|component)$/i
```

3. checkComponentAttr

检查 el 是否为组件，如果是则返回组件 id。

- el {Element}
- options {Object}

```
<!--源码目录：src/util/components.js 37 行-->
function checkComponentAttr (el, options) {
  var tag = el.tagName.toLowerCase()
  var hasAttrs = el.hasAttributes()
  // 如果不是普通元素，不是内置自定义元素
  if (!commonTagRE.test(tag) && !reservedTagRE.test(tag)) {
  // 如果 options['components'][tag](内部会对 tag 做名称转换)存在
    if (resolveAsset(options, 'components', tag)) {
      return { id: tag }
    } else {
      // getIsBinding 获取 el 元素上 is 绑定的组件名称
      var is = hasAttrs && getIsBinding(el, options)
      if (is) {
        return is
      } else if (process.env.NODE_ENV !== 'production') {
        var expectedTag =
          options._componentNameMap &&
          options._componentNameMap[tag]
        if (expectedTag) {
          warn(
            'Unknown custom element: <' + tag + '> - ' +
            'did you mean <' + expectedTag + '>? ' +
            'HTML is case-insensitive, remember to use kebab-case in templates.'
          )
        } else if (isUnknownElement(el, tag)) {
          warn(
            'Unknown custom element: <' + tag + '> - did you ' +
            'register the component correctly? For recursive components, ' +
```

```
            'make sure to provide the "name" option.'
          )
        }
      }
    }
  } else if (hasAttrs) {
    // getIsBinding 获取 el 元素上 is 绑定的组件名称
    return getIsBinding(el, options)
  }
}
```

19.5　options

顾名思义，本节介绍参数相关操作，如图 19-7 所示。

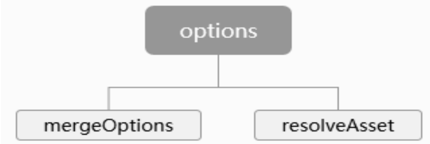


图19-7　options

1. mergeOptions

此方法的核心是 strats 对象，因此我们先来看一下 strats 对象里面有什么，如图 19-8 所示。

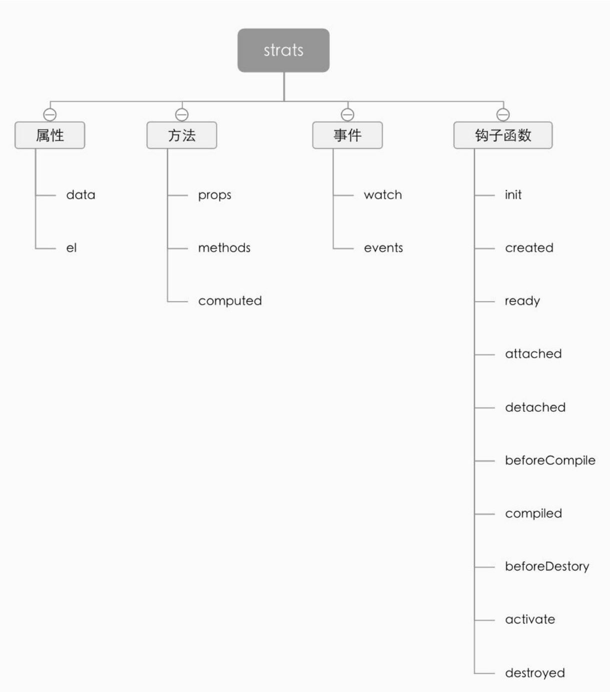


图19-8　strats

接下来我们看看方法都是怎么实现的。

（1）data（见图 19-9）

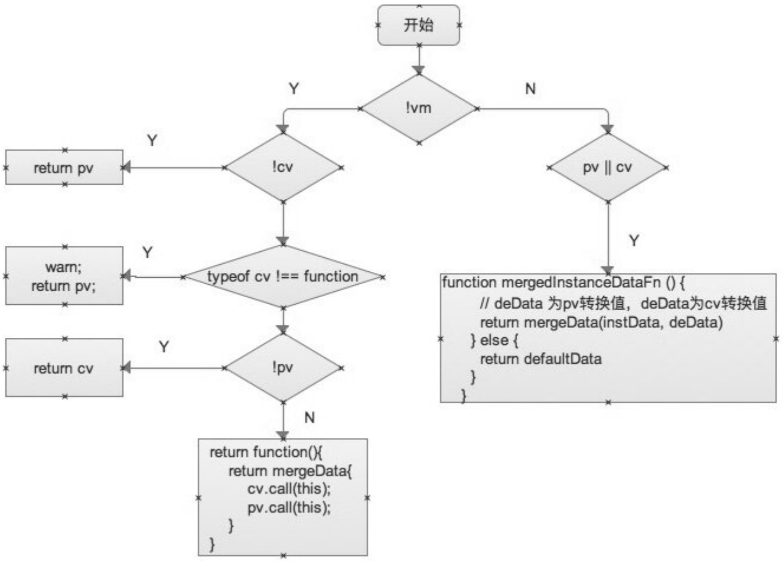


图19-9　data

（2）el（见图 19-10）

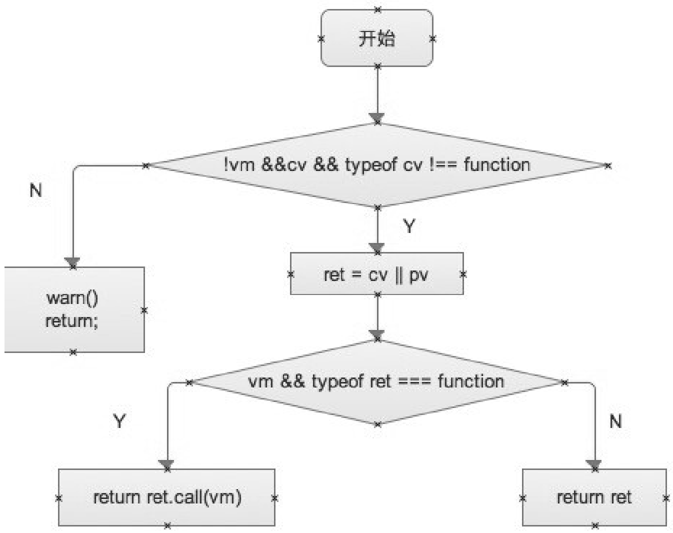


图19-10　el

（3）钩子函数（见图 19-11）

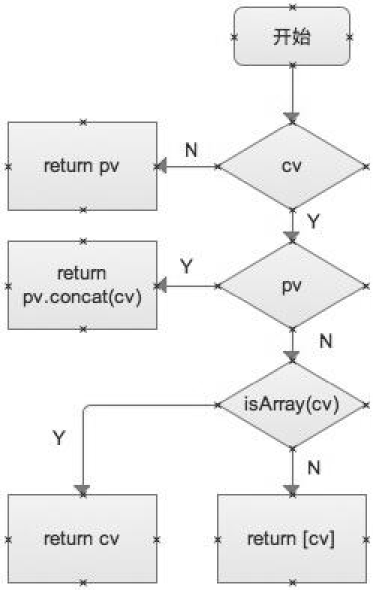


图19-11　钩子函数

（4）props & methods & computed（见图 19-12）

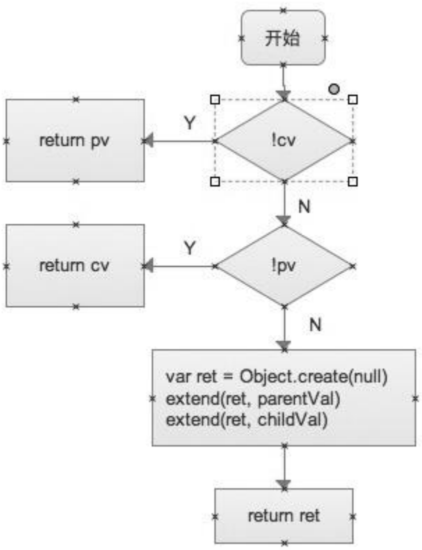


图19-12　prop

（5）watch & events（见图 19-13）

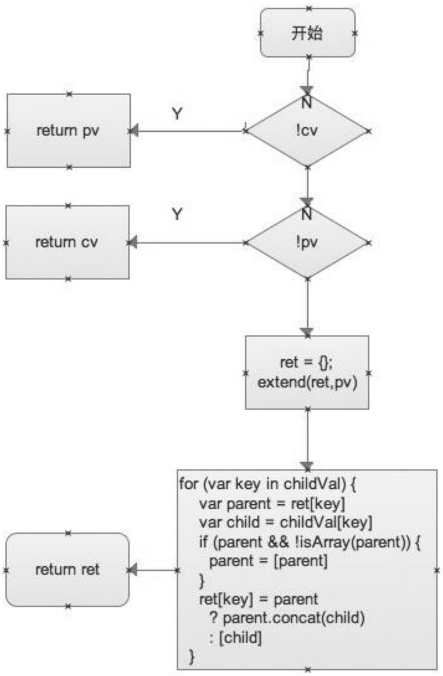


图19-13　watch

看完上面的 starts 对象，我们再看源代码会更清晰一些。

- parent {Object}
- child {Object}
- vm {Vue}

```
<!--源码目录：src/util/components.js 325 行-->
function mergeOptions (parent, child, vm) {
  guardComponents(child)// 确保 child.components 有正确的数据格式
  guardProps(child)// 确保 child.props 被规范化为基于对象的格式
  if (process.env.NODE_ENV !== 'production') {
    if (child.propsData && !vm) {
      warn('propsData can only be used as an instantiation option.')
    }
  }
```

```
  var options = {}
  var key
  //递归
  if (child.extends) {
    parent = typeof child.extends === 'function'
      ? mergeOptions(parent, child.extends.options, vm)
      : mergeOptions(parent, child.extends, vm)
  }
  // 递归
  if (child.mixins) {
    for (var i = 0, l = child.mixins.length; i < l; i++) {
      parent = mergeOptions(parent, child.mixins[i], vm)
    }
  }
  for (key in parent) {
    mergeField(key)
  }
  for (key in child) {
    if (!hasOwn(parent, key)) {
      mergeField(key)
    }
  }
  function mergeField (key) {
    var strat = strats[key] || defaultStrat
    options[key] = strat(parent[key], child[key], vm, key)
  }
  return options
}
```

2. resolveAsset

如果 options[type][id]（内部会对 tag 做名称转换）存在，则返回；否则返回 false。

```
<!--源码目录：src/util/components.js 372 行-->
function resolveAsset (options, type, id, warnMissing) {
  if (typeof id !== 'string') {
    return
  }
  var assets = options[type]
  var camelizedId
```

```
  var res = assets[id] ||
    // camelCase ID
    assets[camelizedId = camelize(id)] ||
    // Pascal Case ID
    assets[camelizedId.charAt(0).toUpperCase() + camelizedId.slice(1)]
  if (process.env.NODE_ENV !== 'production' && warnMissing && !res) {
    warn(
      'Failed to resolve ' + type.slice(0, -1) + ': ' + id,
      options
    )
  }
  return res
}
```

19.6 debug

warn——使用 console.error 输出警告信息。

```
<!--源码目录：src/util/warn.js 7 行-->
let warn
let formatComponentName
if (process.env.NODE_ENV !== 'production') {
  const hasConsole = typeof console !== 'undefined'
  warn = (msg, vm) => {
    if (hasConsole && (!config.silent)) {
      console.error('[Vue warn]: ' + msg + (vm ? formatComponentName(vm) : ''))
    }
  }
  formatComponentName = vm => {
    var name = vm._isVue ? vm.$options.name : vm.name
    return name
      ? ' (found in component: <' + hyphenate(name) + '>)'
      : ''
  }
}
```

第 20 章

源码篇——深入响应式原理

Vue.js 最显著的功能就是响应式系统，它是一个典型的 MVVM 框架，模型（Model）只是普通的 JavaScript 对象，修改它则视图（View）会自动更新。这种设计让状态管理变得非常简单而直观，不过理解它的原理也很重要，可以避免一些常见问题。下面让我们深挖 Vue.js 响应式系统的细节，来看一看 Vue.js 是如何把模型和视图建立起关联关系的。

20.1 如何追踪变化

我们先来看一个简单的例子。代码示例如下：

```
<div id="main">
  <h1>count: {{times}}</h1>
</div>
<script src="vue.js"></script>
<script>
  var vm = new Vue({
    el: '#main',
    data: function () {
      return {
        times: 1
      };
    },
    created: function () {
      var me = this;
      setInterval(function () {
        me.times++;
      }, 1000);
```

```
    }
  });
</script>
```

运行后，我们可以从页面中看到，count 后面的 times 每隔 1s 递增 1，视图一直在更新。在代码中仅仅是通过 setInterval 方法每隔 1s 来修改 vm.times 的值，并没有任何 DOM 操作。那么 Vue.js 是如何实现这个过程的呢？我们可以通过一张图来看一下，如图 20-1 所示。

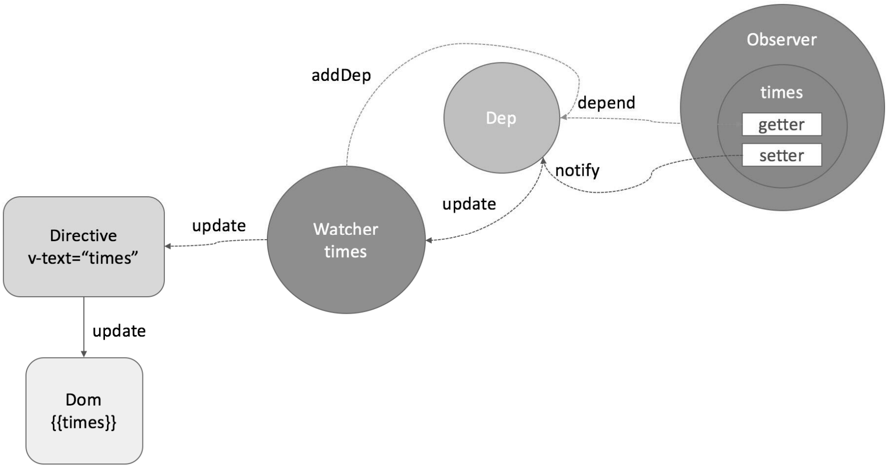


图20-1　模型和视图关联关系图

图中的模型（Model）就是 data 方法返回的{times:1}，视图（View）是最终在浏览器中显示的 DOM。模型通过 Observer、Dep、Watcher、Directive 等一系列对象的关联，最终和视图建立起关系。归纳起来，Vue.js 在这里主要做了三件事：

- 通过 Observer 对 data 做监听，并且提供了订阅某个数据项变化的能力。
- 把 template 编译成一段 document fragment，然后解析其中的 Directive，得到每一个 Directive 所依赖的数据项和 update 方法。
- 通过 Watcher 把上述两部分结合起来，即把 Directive 中的数据依赖通过 Watcher 订阅在对应数据的 Observer 的 Dep 上。当数据变化时，就会触发 Observer 的 Dep 上的 notify 方法通知对应的 Watcher 的 update，进而触发 Directive 的 update 方法来更新 DOM 视图，最后达到模型和视图关联起来。

接下来我们就结合 Vue.js 的源码来详细介绍这三个过程。

20.1.1　Observer

首先来看一下 Vue.js 是如何给 data 对象添加 Observer 的。我们知道，Vue 实例创建的过程会有一个生命周期，其中有一个过程就是调用 vm._initData 方法处理 data 选项。_initData 方法的源码定义如下：

```
<!--源码目录：src/instance/internal/state.js-->
Vue.prototype._initData = function () {
    var dataFn = this.$options.data
    var data = this._data = dataFn ? dataFn() : {}
    if (!isPlainObject(data)) {
      data = {}
      process.env.NODE_ENV !== 'production' && warn(
        'data functions should return an object.',
        this
      )
    }
    var props = this._props
    // proxy data on instance
    var keys = Object.keys(data)
    var i, key
    i = keys.length
    while (i--) {
      key = keys[i]
      // there are two scenarios where we can proxy a data key:
      // 1. it's not already defined as a prop
      // 2. it's provided via a instantiation option AND there are no
      // template prop present
      if (!props || !hasOwn(props, key)) {
        this._proxy(key)
      } else if (process.env.NODE_ENV !== 'production') {
        warn(
          'Data field "' + key + '" is already defined ' +
          'as a prop. To provide default value for a prop, use the "default" ' +
          'prop option; if you want to pass prop values to an instantiation ' +
          'call, use the "propsData" option.',
          this
        )
      }
```

```
    }
    // observe data
    observe(data, this)
  }
```

在_initData 中我们要特别注意_proxy 方法，它的功能就是遍历 data 的 key，把 data 上的属性代理到 vm 实例上。_proxy 方法的源码定义如下：

```
<!--源码目录：src/instance/internal/state.js-->
Vue.prototype._proxy = function (key) {
    if (!isReserved(key)) {
      // need to store ref to self here
      // because these getter/setters might
      // be called by child scopes via
      // prototype inheritance.
      var self = this
      Object.defineProperty(self, key, {
        configurable: true,
        enumerable: true,
        get: function proxyGetter () {
          return self._data[key]
        },
        set: function proxySetter (val) {
          self._data[key] = val
        }
      })
    }
  }
```

_proxy 方法主要通过 Object.defineProperty 的 getter 和 setter 方法实现了代理。在前面的例子中，我们调用 vm.times 就相当于访问了 vm._data.times。

在_initData 方法的最后，我们调用了 observe(data, this)方法来对 data 做监听。observe 方法的源码定义如下：

```
<!--源码目录：src/observer/index.js-->
export function observe (value, vm) {
  if (!value || typeof value !== 'object') {
    return
  }
  var ob
```

```
  if (
    hasOwn(value, '__ob__') &&
    value.__ob__ instanceof Observer
  ) {
    ob = value.__ob__
  } else if (
    shouldConvert &&
    (isArray(value) || isPlainObject(value)) &&
    Object.isExtensible(value) &&
    !value._isVue
  ) {
    ob = new Observer(value)
  }
  if (ob && vm) {
    ob.addVm(vm)
  }
  return ob
}
```

observe 方法首先判断 value 是否已经添加了__ob__属性，它是一个 Observer 对象的实例。如果是就直接用，否则在 value 满足一些条件（数组或对象、可扩展、非 vue 组件等）的情况下创建一个 Observer 对象。接下来我们看一下 Observer 这个类，它的源码定义如下：

```
<!--源码目录：src/observer/index.js-->
export function Observer (value) {
  this.value = value
  this.dep = new Dep()
  def(value, '__ob__', this)
  if (isArray(value)) {
    var augment = hasProto
      ? protoAugment
      : copyAugment
    augment(value, arrayMethods, arrayKeys)
    this.observeArray(value)
  } else {
    this.walk(value)
  }
}
```

Observer 类的构造函数主要做了这么几件事：首先创建了一个 Dep 对象实例（关于 Dep 对象我们稍后作介绍）；然后把自身 this 添加到 value 的__ob__属性上；最后对 value 的类型进行

判断，如果是数组则观察数组，否则观察单个元素。其实 observeArray 方法就是对数组进行遍历，递归调用 observe 方法，最终都会调用 walk 方法观察单个元素。接下来我们看一下 walk 方法，它的源码定义如下：

```
<!--源码目录：src/observer/index.js-->
Observer.prototype.walk = function (obj) {
  var keys = Object.keys(obj)
  for (var i = 0, l = keys.length; i < l; i++) {
    this.convert(keys[i], obj[keys[i]])
  }
}
```

walk 方法是对 obj 的 key 进行遍历，依次调用 convert 方法，对 obj 的每一个属性进行转换，让它们拥有 getter、setter 方法。只有当 obj 是一个对象时，这个方法才能被调用。接下来我们看一下 convert 方法，它的源码定义如下：

```
<!--源码目录：src/observer/index.js-->
Observer.prototype.convert = function (key, val) {
  defineReactive(this.value, key, val)
}
```

convert 方法很简单，它调用了 defineReactive 方法。这里 this.value 就是要观察的 data 对象，key 是 data 对象的某个属性，val 则是这个属性的值。defineReactive 的功能是把要观察的 data 对象的每个属性都赋予 getter 和 setter 方法。这样一旦属性被访问或者更新，我们就可以追踪到这些变化。接下来我们看一下 defineReactive 方法，它的源码定义如下：

```
<!--源码目录：src/observer/index.js-->
export function defineReactive (obj, key, val) {
  var dep = new Dep()
  var property = Object.getOwnPropertyDescriptor(obj, key)
  if (property && property.configurable === false) {
    return
  }
  // cater for pre-defined getter/setters
  var getter = property && property.get
  var setter = property && property.set
  var childOb = observe(val)
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter () {
```

```
      var value = getter ? getter.call(obj) : val
      if (Dep.target) {
        dep.depend()
        if (childOb) {
          childOb.dep.depend()
        }
        if (isArray(value)) {
          for (var e, i = 0, l = value.length; i < l; i++) {
            e = value[i]
            e && e.__ob__ && e.__ob__.dep.depend()
          }
        }
      }
      return value
    },
    set: function reactiveSetter (newVal) {
      var value = getter ? getter.call(obj) : val
      if (newVal === value) {
        return
      }
      if (setter) {
        setter.call(obj, newVal)
      } else {
        val = newVal
      }
      childOb = observe(newVal)
      dep.notify()
    }
  })
}
```

defineReactive 方法最核心的部分就是通过调用 Object.defineProperty 给 data 的每个属性添加 getter 和 setter 方法。当 data 的某个属性被访问时，则会调用 getter 方法，判断当 Dep.target 不为空时调用 dep.depend 和 childObj.dep.depend 方法做依赖收集。如果访问的属性是一个数组，则会遍历这个数组收集数组元素的依赖。当改变 data 的属性时，则会调用 setter 方法，这时调用 dep.notify 方法进行通知。这里我们提到了 dep，它是 Dep 对象的实例。接下来我们看一下 Dep 这个类，它的源码定义如下：

```
<!--源码目录：src/observer/dep.js-->
export default function Dep () {
```

```
  this.id = uid++
  this.subs = []
}
// the current target watcher being evaluated.
// this is globally unique because there could be only one
// watcher being evaluated at any time.
Dep.target = null
```

Dep 类是一个简单的观察者模式的实现。它的构造函数非常简单，初始化了 id 和 subs。其中 subs 用来存储所有订阅它的 Watcher，Watcher 的实现稍后我们会介绍。Dep.target 表示当前正在计算的 Watcher，它是全局唯一的，因为在同一时间只能有一个 Watcher 被计算。

前面提到了在 getter 和 setter 方法调用时会分别调用 dep.depend 方法和 dep.notify 方法，接下来依次介绍这两个方法。depend 方法的源码定义如下：

```
<!--源码目录：src/observer/dep.js-->
Dep.prototype.depend = function () {
  Dep.target.addDep(this)
}
```

depend 方法很简单，它通过 Dep.target.addDep(this)方法把当前 Dep 的实例添加到当前正在计算的 Watcher 的依赖中。接下来我们看一下 notify 方法，它的源码定义如下：

```
<!--源码目录：src/observer/dep.js-->
Dep.prototype.notify = function () {
  // stablize the subscriber list first
  var subs = toArray(this.subs)
  for (var i = 0, l = subs.length; i < l; i++) {
    subs[i].update()
  }
}
```

notify 方法也很简单，它遍历了所有的订阅 Watcher，调用它们的 update 方法。

至此，vm 实例中给 data 对象添加 Observer 的过程就结束了。接下来我们看一下 Vue.js 是如何进行指令解析的。

20.1.2　Directive

Vue 指令类型很多，限于篇幅，我们不会把所有指令的解析过程都介绍一遍，这里结合前面的例子只介绍 v-text 指令的解析过程，其他指令的解析过程也大同小异。

前面我们提到了 Vue 实例创建的生命周期，在给 data 添加 Observer 之后，有一个过程是调用 vm._compile 方法对模板进行编译。_compile 方法的源码定义如下：

```
<!--源码目录：src/instance/internal/lifecycle.js-->
Vue.prototype._compile = function (el) {
    var options = this.$options
    // transclude and init element
    // transclude can potentially replace original
    // so we need to keep reference; this step also injects
    // the template and caches the original attributes
    // on the container node and replacer node.
    var original = el
    el = transclude(el, options)
    this._initElement(el)
    // handle v-pre on root node (#2026)
    if (el.nodeType === 1 && getAttr(el, 'v-pre') !== null) {
      return
    }
    // root is always compiled per-instance, because
    // container attrs and props can be different every time.
    var contextOptions = this._context && this._context.$options
    var rootLinker = compileRoot(el, options, contextOptions)
    // resolve slot distribution
    resolveSlots(this, options._content)
    // compile and link the rest
    var contentLinkFn
    var ctor = this.constructor
    // component compilation can be cached
    // as long as it's not using inline-template
    if (options._linkerCachable) {
      contentLinkFn = ctor.linker
      if (!contentLinkFn) {
        contentLinkFn = ctor.linker = compile(el, options)
      }
    }
    // link phase
    // make sure to link root with prop scope!
    var rootUnlinkFn = rootLinker(this, el, this._scope)
    var contentUnlinkFn = contentLinkFn
```

```
    ? contentLinkFn(this, el)
    : compile(el, options)(this, el)
  // register composite unlink function
  // to be called during instance destruction
  this._unlinkFn = function () {
    rootUnlinkFn()
    // passing destroying: true to avoid searching and
    // splicing the directives
    contentUnlinkFn(true)
  }
  // finally replace original
  if (options.replace) {
    replace(original, el)
  }
  this._isCompiled = true
  this._callHook('compiled')
}
```

我们可以通过图 20-2 来看一下这个方法编译的主要流程。

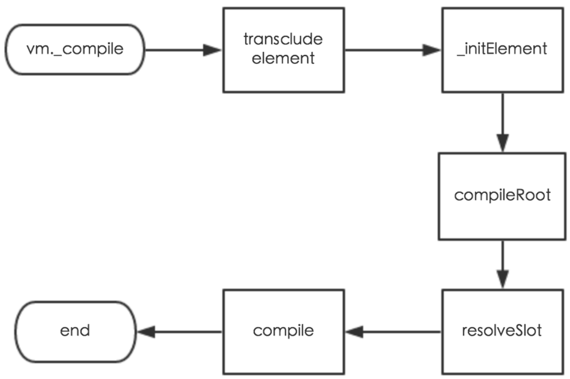


图20-2　vm._compile编译主要流程图

这个过程通过 el = transclude(el, option)方法把 template 编译成一段 document fragment，拿到 el 对象。而指令解析部分就是通过 compile(el, options)方法实现的。接下来我们看一下 compile 方法的实现，它的源码定义如下：

```
<!--源码目录：src/compiler/compile.js-->
export function compile (el, options, partial) {
  // link function for the node itself.
  var nodeLinkFn = partial || !options._asComponent
```

```
    ? compileNode(el, options)
    : null
  // link function for the childNodes
  var childLinkFn =
    !(nodeLinkFn && nodeLinkFn.terminal) &&
    !isScript(el) &&
    el.hasChildNodes()
      ? compileNodeList(el.childNodes, options)
      : null
  /**
   * A composite linker function to be called on a already
   * compiled piece of DOM, which instantiates all directive
   * instances.
   *
   * @param {Vue} vm
   * @param {Element|DocumentFragment} el
   * @param {Vue} [host] - host vm of transcluded content
   * @param {Object} [scope] - v-for scope
   * @param {Fragment} [frag] - link context fragment
   * @return {Function|undefined}
   */
  return function compositeLinkFn (vm, el, host, scope, frag) {
    // cache childNodes before linking parent, fix #657
    var childNodes = toArray(el.childNodes)
    // link
    var dirs = linkAndCapture(function compositeLinkCapturer () {
      if (nodeLinkFn) nodeLinkFn(vm, el, host, scope, frag)
      if (childLinkFn) childLinkFn(vm, childNodes, host, scope, frag)
    }, vm)
    return makeUnlinkFn(vm, dirs)
  }
}
```

compile 方法主要通过 compileNode(el, options)方法完成节点的解析，如果节点拥有子节点，则调用 compileNodeList(el.childNodes, options)方法完成子节点的解析。compileNodeList 方法其实就是遍历子节点，递归调用 compileNode 方法。因为 DOM 元素本身就是树结构，这种递归方法也就是常见的树的深度遍历方法，这样就可以完成整个 DOM 树节点的解析。接下来我们看一下 compileNode 方法的实现，它的源码定义如下：

```
<!--源码目录：src/compiler/compile.js-->
function compileNode (node, options) {
```

```
  var type = node.nodeType
  if (type === 1 && !isScript(node)) {
    return compileElement(node, options)
  } else if (type === 3 && node.data.trim()) {
    return compileTextNode(node, options)
  } else {
    return null
  }
}
```

compileNode 方法对节点的 nodeType 做判断，如果是一个非 script 普通的元素（div、p 等）；则调用 compileElement(node, options)方法解析；如果是一个非空的文本节点，则调用 compileTextNode(node, options)方法解析。我们在前面的例子中解析的是非空文本节点 count: {{times}}，这实际上是 v-text 指令，它的解析是通过 compileTextNode 方法实现的。接下来我们看一下 compileTextNode 方法，它的源码定义如下：

```
<!--源码目录：src/compiler/compile.js-->
function compileTextNode (node, options) {
  // skip marked text nodes
  if (node._skip) {
    return removeText
  }
  var tokens = parseText(node.wholeText)
  if (!tokens) {
    return null
  }
  // mark adjacent text nodes as skipped,
  // because we are using node.wholeText to compile
  // all adjacent text nodes together. This fixes
  // issues in IE where sometimes it splits up a single
  // text node into multiple ones.
  var next = node.nextSibling
  while (next && next.nodeType === 3) {
    next._skip = true
    next = next.nextSibling
  }
  var frag = document.createDocumentFragment()
  var el, token
  for (var i = 0, l = tokens.length; i < l; i++) {
```

```
    token = tokens[i]
    el = token.tag
      ? processTextToken(token, options)
      : document.createTextNode(token.value)
    frag.appendChild(el)
  }
  return makeTextNodeLinkFn(tokens, frag, options)
}
```

compileTextNode 方法首先调用了 parseText 方法对 node.wholeText 做解析。主要通过正则表达式解析 count: {{times}}部分，我们看一下解析结果，如图 20-3 所示。

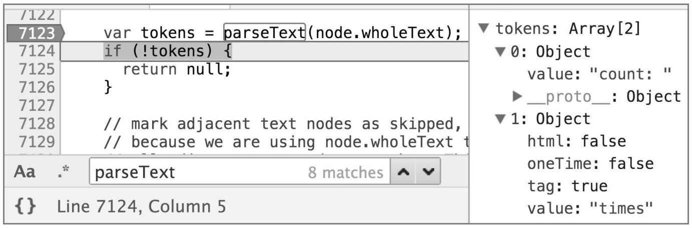


图20-3　parseText解析文本节点结果

解析后的 tokens 是一个数组，数组的每个元素则是一个 Object。如果是 count: 这样的普通文本，则返回的对象只有 value 字段；如果是{{times}}这样的插值，则返回的对象包含 html、onTime、tag、value 等字段。

接下来创建 document fragment，遍历 tokens 创建 DOM 节点插入到这个 fragment 中。在遍历过程中，如果 token 无 tag 字段，则调用 document.createTextNode(token.value)方法创建 DOM 节点；否则调用 processTextToken(token, options)方法创建 DOM 节点和扩展 token 对象。我们看一下调用后的结果，如图 20-4 所示。

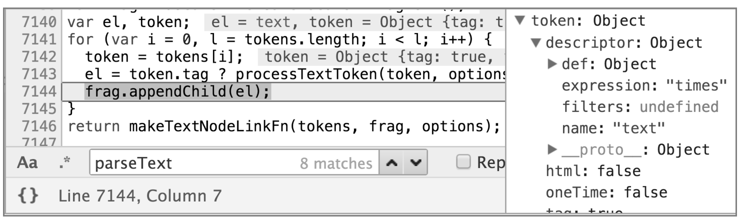


图20-4　processTextToken解析文本节点结果

可以看到，token 字段多了一个 descriptor 属性。这个属性包含了几个字段，其中 def 表示指令相关操作的对象，expression 为解析后的表达式，filters 为过滤器，name 为指令的名称。

在 compileTextNode 方法的最后，调用 makeTextNodeLinkFn(tokens, frag, options)并返回该方法执行的结果。接下来我们看一下 makeTextNodeLinkFn 方法，它的源码定义如下：

```
<!--源码目录：src/compiler/compile.js-->
function makeTextNodeLinkFn (tokens, frag) {
  return function textNodeLinkFn (vm, el, host, scope) {
    var fragClone = frag.cloneNode(true)
    var childNodes = toArray(fragClone.childNodes)
    var token, value, node
    for (var i = 0, l = tokens.length; i < l; i++) {
      token = tokens[i]
      value = token.value
      if (token.tag) {
        node = childNodes[i]
        if (token.oneTime) {
          value = (scope || vm).$eval(value)
          if (token.html) {
            replace(node, parseTemplate(value, true))
          } else {
            node.data = _toString(value)
          }
        } else {
          vm._bindDir(token.descriptor, node, host, scope)
        }
      }
    }
    replace(el, fragClone)
  }
}
```

makeTextNodeLinkFn 这个方法什么也没做，它仅仅是返回了一个新的方法 textNodeLinkFn。往前回溯，这个方法最终作为 compileNode 的返回值，被添加到 compile 方法生成的 childLinkFn 中。

我们回到 compile 方法，在 compile 方法的最后有这样一段代码：

```
<!--源码目录：src/compiler/compile.js-->
return function compositeLinkFn (vm, el, host, scope, frag) {
  // cache childNodes before linking parent, fix #657
  var childNodes = toArray(el.childNodes)
```

```
    // link
    var dirs = linkAndCapture(function compositeLinkCapturer () {
      if (nodeLinkFn) nodeLinkFn(vm, el, host, scope, frag)
      if (childLinkFn) childLinkFn(vm, childNodes, host, scope, frag)
    }, vm)
    return makeUnlinkFn(vm, dirs)
  }
```

compile 方法返回了 compositeLinkFn，它在 Vue.prototype._compile 方法执行时，是通过 compile(el, options)(this, el)调用的。compositeLinkFn 方法执行了 linkAndCapture 方法，它的功能是通过调用 compile 过程中生成的 link 方法创建指令对象，再对指令对象做一些绑定操作。linkAndCapture 方法的源码定义如下：

```
<!--源码目录：src/compiler/compile.js-->
function linkAndCapture (linker, vm) {
  /* istanbul ignore if */
  if (process.env.NODE_ENV === 'production') {
    // reset directives before every capture in production
    // mode, so that when unlinking we don't need to splice
    // them out (which turns out to be a perf hit).
    // they are kept in development mode because they are
    // useful for Vue's own tests.
    vm._directives = []
  }
  var originalDirCount = vm._directives.length
  linker()
  var dirs = vm._directives.slice(originalDirCount)
  dirs.sort(directiveComparator)
  for (var i = 0, l = dirs.length; i < l; i++) {
    dirs[i]._bind()
  }
  return dirs
}
```

linkAndCapture 方法首先调用了 linker 方法，它会遍历 compile 过程中生成的所有 linkFn 并调用，本例中会调用到之前定义的 textNodeLinkFn。这个方法会遍历 tokens，判断如果 token 的 tag 属性值为 true 且 oneTime 属性值为 false，则调用 vm._bindDir(token.descriptor, node, host, scope) 方法创建指令对象。vm._bindDir 方法的源码定义如下：

```
<!--源码目录：src/instance/internal/lifecycle.js-->
```

```
Vue.prototype._bindDir = function (descriptor, node, host, scope, frag) {
    this._directives.push(
      new Directive(descriptor, this, node, host, scope, frag)
    )
}
```

Vue.prototype._bindDir 方法就是根据 descriptor 实例化不同的 Directive 对象，并添加到 vm 实例的 directives 数组中的。到这一步，Vue.js 从解析模板到生成 Directive 对象的步骤就完成了。接下来回到 linkAndCapture 方法，它对创建好的 directives 进行排序，然后遍历 directives 调用 dirs[i]._bind 方法对单个 directive 做一些绑定操作。dirs[i]._bind 方法的源码定义如下：

```
<!--源码目录：src/directive.js-->
Directive.prototype._bind = function () {
  var name = this.name
  var descriptor = this.descriptor
  // remove attribute
  if (
    (name !== 'cloak' || this.vm._isCompiled) &&
    this.el && this.el.removeAttribute
  ) {
    var attr = descriptor.attr || ('v-' + name)
    this.el.removeAttribute(attr)
  }
  // copy def properties
  var def = descriptor.def
  if (typeof def === 'function') {
    this.update = def
  } else {
    extend(this, def)
  }
  // setup directive params
  this._setupParams()
  // initial bind
  if (this.bind) {
    this.bind()
  }
  this._bound = true
  if (this.literal) {
    this.update && this.update(descriptor.raw)
  } else if (
```

```
    (this.expression || this.modifiers) &&
    (this.update || this.twoWay) &&
    !this._checkStatement()
  ) {
    // wrapped updater for context
    var dir = this
    if (this.update) {
      this._update = function (val, oldVal) {
        if (!dir._locked) {
          dir.update(val, oldVal)
        }
      }
    } else {
      this._update = noop
    }
    var preProcess = this._preProcess
      ? bind(this._preProcess, this)
      : null
    var postProcess = this._postProcess
      ? bind(this._postProcess, this)
      : null
    var watcher = this._watcher = new Watcher(
      this.vm,
      this.expression,
      this._update, // callback
      {
        filters: this.filters,
        twoWay: this.twoWay,
        deep: this.deep,
        preProcess: preProcess,
        postProcess: postProcess,
        scope: this._scope
      }
    )
    // v-model with inital inline value need to sync back to
    // model instead of update to DOM on init. They would
    // set the afterBind hook to indicate that.
    if (this.afterBind) {
      this.afterBind()
    } else if (this.update) {
```

```
      this.update(watcher.value)
    }
  }
}
```

Directive.prototype._bind 方法的主要功能就是做一些指令的初始化操作，如混合 def 属性。def 是通过 this.descriptor.def 获得的，this.descriptor 是对指令进行相关描述的对象，而 this.descriptor.def 则是包含指令相关操作的对象。比如对于 v-text 指令，我们可以看一下它的相关操作，源码定义如下：

```
<!--源码目录：src/directives/public/text.js-->
export default {
  bind () {
    this.attr = this.el.nodeType === 3
      ? 'data'
      : 'textContent'
  },
  update (value) {
    this.el[this.attr] = _toString(value)
  }
}
```

v-text 的 def 包含了 bind 和 update 方法，Directive 在初始化时通过 extend(this, def)方法可以对实例扩展这两个方法。Directive 在初始化时还定义了 this._update 方法，并创建了 Watcher，把 this._update 方法作为 Watcher 的回调函数。这里把 Directive 和 Watcher 做了关联，当 Watcher 观察到指令表达式值变化时，会调用 Directive 实例的_update 方法，最终调用 v-text 的 update 方法更新 DOM 节点。

至此，vm 实例中编译模板、解析指令、绑定 Watcher 的过程就结束了。接下来我们看一下 Watcher 的实现，了解 Directive 和 Observer 之间是如何通过 Watcher 关联的。

20.1.3　Watcher

我们先来看一下 Watcher 类的实现，它的源码定义如下：

```
<!--源码目录：src/watcher.js-->
export default function Watcher (vm, expOrFn, cb, options) {
  // mix in options
  if (options) {
    extend(this, options)
```

```
  }
  var isFn = typeof expOrFn === 'function'
  this.vm = vm
  vm._watchers.push(this)
  this.expression = expOrFn
  this.cb = cb
  this.id = ++uid // uid for batching
  this.active = true
  this.dirty = this.lazy // for lazy watchers
  this.deps = []
  this.newDeps = []
  this.depIds = new Set()
  this.newDepIds = new Set()
  this.prevError = null // for async error stacks
  // parse expression for getter/setter
  if (isFn) {
    this.getter = expOrFn
    this.setter = undefined
  } else {
    var res = parseExpression(expOrFn, this.twoWay)
    this.getter = res.get
    this.setter = res.set
  }
  this.value = this.lazy
    ? undefined
    : this.get()
  // state for avoiding false triggers for deep and Array
  // watchers during vm._digest()
  this.queued = this.shallow = false
}
```

Directive 实例在初始化 Watcher 时，会传入指令的 expression。Watcher 构造函数会通过 parseExpression(expOrFn, this.twoWay)方法对 expression 做进一步的解析。在前面的例子中，expression 是 times，passExpression 方法的功能是把 expression 转换成一个对象，如图 20-5 所示。

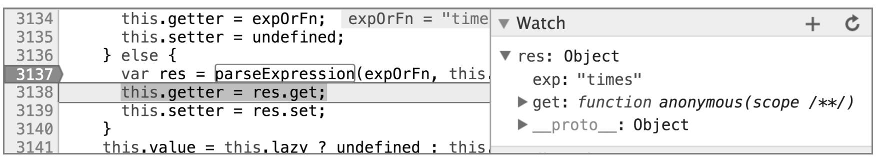


图20-5　passExpression执行结果

可以看到 res 有两个属性，其中 exp 为表达式字符串；get 是通过 new Function 生成的匿名方法，可以把它打印出来，如图 20-6 所示。

```
> res.get.toString()
<· "function anonymous(scope
  /**/) {
  return scope.times;
  }"
```

图20-6　res.get方法打印结果

可以看到 res.get 方法很简单，它接受传入一个 scope 变量，返回 scope.times。对于传入的 scope 值，稍后我们会进行介绍。在 Watcher 构造函数的最后调用了 this.get 方法，它的源码定义如下：

```
<!--源码目录：src/watcher.js-->
Watcher.prototype.get = function () {
  this.beforeGet()
  var scope = this.scope || this.vm
  var value
  try {
    value = this.getter.call(scope, scope)
  } catch (e) {
    if (
      process.env.NODE_ENV !== 'production' &&
      config.warnExpressionErrors
    ) {
      warn(
        'Error when evaluating expression ' +
        '"' + this.expression + '": ' + e.toString(),
        this.vm
      )
    }
  }
  // "touch" every property so they are all tracked as
  // dependencies for deep watching
  if (this.deep) {
    traverse(value)
  }
  if (this.preProcess) {
    value = this.preProcess(value)
  }
```

```
  if (this.filters) {
    value = scope._applyFilters(value, null, this.filters, false)
  }
  if (this.postProcess) {
    value = this.postProcess(value)
  }
  this.afterGet()
  return value
}
```

Watcher.prototype.get 方法的功能就是对当前 Watcher 进行求值，收集依赖关系。它首先执行 this.beforeGet 方法，源码定义如下：

```
<!--源码目录：src/watcher.js-->
Watcher.prototype.beforeGet = function () {
  Dep.target = this
}
```

Watcher.prototype.beforeGet 很简单，设置 Dep.target 为当前 Watcher 实例，为接下来的依赖收集做准备。我们回到 get 方法，接下来执行 this.getter.call(scope, scope)方法，这里的 scope 是 this.vm，也就是当前 Vue 实例。这个方法实际上相当于获取 vm.times，这样就触发了对象的 getter。在 20.1.1 节我们给 data 添加 Observer 时，通过 Object.defineProperty 给 data 对象的每一个属性添加 getter 和 setter。回顾一下代码：

```
<!--源码目录：src/observer/index.js-->
Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter () {
      var value = getter ? getter.call(obj) : val
      if (Dep.target) {
        dep.depend()
        if (childOb) {
          childOb.dep.depend()
        }
        if (isArray(value)) {
          for (var e, i = 0, l = value.length; i < l; i++) {
            e = value[i]
            e && e.__ob__ && e.__ob__.dep.depend()
          }
        }
```

```
      }
      return value
    },
    …
  })
```

当获取 vm.times 时，会执行到 get 方法体内。由于我们在之前已经设置了 Dep.target 为当前 Watcher 实例，所以接下来就调用 dep.depend()方法完成依赖收集。它实际上是执行了 Dep.target.addDep(this)，相当于执行了 Watcher 实例的 addDep 方法，把 Dep 实例添加到 Watcher 实例的依赖中。addDep 方法的源码定义如下：

```
<!--源码目录：src/watcher.js-->
Watcher.prototype.addDep = function (dep) {
  var id = dep.id
  if (!this.newDepIds.has(id)) {
    this.newDepIds.add(id)
    this.newDeps.push(dep)
    if (!this.depIds.has(id)) {
      dep.addSub(this)
    }
  }
}
```

Watcher.prototype.addDep 方法就是把 dep 添加到 Watcher 实例的依赖中，同时又通过 dep.addSub(this)把 Watcher 实例添加到 dep 的订阅者中。addSub 方法的源码定义如下：

```
<!--源码目录：src/observer/dep.js-->
Dep.prototype.addSub = function (sub) {
  this.subs.push(sub)
}
```

至此，指令完成了依赖收集，并且通过 Watcher 完成了对数据变化的订阅。

接下来我们看一下，当 data 发生变化时，视图是如何自动更新的。在前面的例子中，我们通过 setInterval 每隔 1s 执行一次 vm.times++，数据改变会触发对象的 setter，执行 set 方法体的代码。回顾一下代码：

```
<!--源码目录：src/observer/index.js-->
Object.defineProperty(obj, key, {
    …
    set: function reactiveSetter (newVal) {
      var value = getter ? getter.call(obj) : val
```

```
      if (newVal === value) {
        return
      }
      if (setter) {
        setter.call(obj, newVal)
      } else {
        val = newVal
      }
      childOb = observe(newVal)
      dep.notify()
    }
})
```

这里会调用 dep.notify()方法，它会遍历所有的订阅者，也就是 Watcher 实例。然后调用 Watcher 实例的 update 方法，源码定义如下：

```
<!--源码目录：src/watcher.js-->
Watcher.prototype.update = function (shallow) {
  if (this.lazy) {
    this.dirty = true
  } else if (this.sync || !config.async) {
    this.run()
  } else {
    // if queued, only overwrite shallow with non-shallow,
    // but not the other way around.
    this.shallow = this.queued
      ? shallow
        ? this.shallow
        : false
      : !!shallow
    this.queued = true
    // record before-push error stack in debug mode
    /* istanbul ignore if */
    if (process.env.NODE_ENV !== 'production' && config.debug) {
      this.prevError = new Error('[vue] async stack trace')
    }
    pushWatcher(this)
  }
}
```

Watcher.prototype.update 方法在满足某些条件下会直接调用 this.run 方法。在多数情况下会

调用 pushWatcher(this)方法把 Watcher 实例推入队列中，延迟 this.run 调用的时机。pushWatcher 方法的源码定义如下：

```
<!--源码目录：src/batcher.js-->
export function pushWatcher (watcher) {
  const id = watcher.id
  if (has[id] == null) {
    // push watcher into appropriate queue
    const q = watcher.user
      ? userQueue
      : queue
    has[id] = q.length
    q.push(watcher)
    // queue the flush
    if (!waiting) {
      waiting = true
      nextTick(flushBatcherQueue)
    }
  }
}
```

pushWatcher 方法把 Watcher 推入队列中，通过 nextTick 方法在下一个事件循环周期处理 Watcher 队列，这是 Vue.js 的一种性能优化手段。因为如果同时观察的数据多次变化，比如同步执行 3 次 vm.time++，同步调用 watcher.run 就会触发 3 次 DOM 操作。而推入队列中等待下一个事件循环周期再操作队列里的 Watcher，因为是同一个 Watcher，它只会调用一次 watcher.run，从而只触发一次 DOM 操作。接下来我们看一下 flushBatcherQueue 方法，它的源码定义如下：

```
<!--源码目录：src/batcher.js-->
function flushBatcherQueue () {
  runBatcherQueue(queue)
  runBatcherQueue(userQueue)
  // user watchers triggered more watchers,
  // keep flushing until it depletes
  if (queue.length) {
    return flushBatcherQueue()
  }
  // dev tool hook
  /* istanbul ignore if */
  if (devtools && config.devtools) {
    devtools.emit('flush')
```

```
  }
  resetBatcherState()
}
```

flushBatcherQueue 方法通过调用 runBatcherQueue 来 run Watcher。这里我们看到 Watcher 队列分为内部 queue 和 userQueue，其中 userQueue 是通过$watch()方法注册的 Watcher。我们优先 run 内部 queue 来保证指令和 DOM 节点优先更新，这样当用户自定义的 Watcher 的回调函数触发时 DOM 已更新完毕。接下来我们看一下 runBatcherQueue 方法，它的源码定义如下：

```
<!--源码目录：src/batcher.js-->
function runBatcherQueue (queue) {
  // do not cache length because more watchers might be pushed
  // as we run existing watchers
  for (let i = 0; i < queue.length; i++) {
    var watcher = queue[i]
    var id = watcher.id
    has[id] = null
    watcher.run()
    // in dev build, check and stop circular updates.
    if (process.env.NODE_ENV !== 'production' && has[id] != null) {
      circular[id] = (circular[id] || 0) + 1
      if (circular[id] > config._maxUpdateCount) {
        warn(
          'You may have an infinite update loop for watcher ' +
          'with expression "' + watcher.expression + '"',
          watcher.vm
        )
        break
      }
    }
  }
  queue.length = 0
}
```

runBatcherQueued 的功能就是遍历 queue 中 Watcher 的 run 方法。接下来我们看一下 Watcher 的 run 方法，它的源码定义如下：

```
<!--源码目录：src/watcher.js-->
Watcher.prototype.run = function () {
  if (this.active) {
    var value = this.get()
```

```
    if (
      value !== this.value ||
      // Deep watchers and watchers on Object/Arrays should fire even
      // when the value is the same, because the value may
      // have mutated; but only do so if this is a
      // non-shallow update (caused by a vm digest).
      ((isObject(value) || this.deep) && !this.shallow)
    ) {
      // set new value
      var oldValue = this.value
      this.value = value
      // in debug + async mode, when a watcher callbacks
      // throws, we also throw the saved before-push error
      // so the full cross-tick stack trace is available.
      var prevError = this.prevError
      /* istanbul ignore if */
      if (process.env.NODE_ENV !== 'production' &&
          config.debug && prevError) {
        this.prevError = null
        try {
          this.cb.call(this.vm, value, oldValue)
        } catch (e) {
          nextTick(function () {
            throw prevError
          }, 0)
          throw e
        }
      } else {
        this.cb.call(this.vm, value, oldValue)
      }
    }
    this.queued = this.shallow = false
  }
}
```

Watcher.prototype.run 方法再次对 Watcher 求值，重新收集依赖。接下来判断求值结果和之前 value 的关系。如果不变则什么也不做，如果变了则调用 this.cb.call(this.vm, value, oldValue) 方法。这个方法是 Directive 实例创建 Watcher 时传入的，它对应相关指令的 update 方法来真实更新 DOM。这样就完成了数据更新到对应视图的变化过程。

Watcher 巧妙地把 Observer 和 Directive 关联起来，实现了数据一旦更新，视图就会自动变化的效果。尽管 Vue.js 利用 Object.defineProperty 这个核心技术实现了数据和视图的绑定，但仍然会存在一些数据变化检测不到的问题，接下来我们看一下这部分内容。

20.2　变化检测问题

受 ES 5 的限制，Vue.js 不能检测到对象属性的添加和删除。因为 Vue.js 在初始化实例时将属性转换为 getter/setter，所以属性必须在 data 对象上已存在才能让 Vue.js 转换它，才能让它是响应式的。代码示例如下：

```
<div id="main">
  <h1>{{a}} {{b}}</h1>
</div>
<script src="vue.js"></script>
<script>
  var vm = new Vue({
    el: '#main',
    data: {a: 1}
  });
  // vm.a 现在是响应式的

  vm.b = 2;
  // vm.b 不是响应式的
</script>
```

我们发现，通过 vm.b=2 给 data 对象添加属性并不会触发视图的变化。不过，有办法在 Vue 实例创建完成之后添加属性并且让它是响应式的。对于 Vue 实例，可以使用$set(key, value)实例方法。代码示例如下：

```
vm.$set('b', 2);
// vm.b 是响应式的
```

为何通过 Vue 实例的$set 方法就能让 vm.b 变成响应式的？我们来一探究竟。$set 方法的源码定义如下：

```
<!--源码目录：src/instance/api/data.js-->
Vue.prototype.$set = function (exp, val) {
   var res = parseExpression(exp, true)
   if (res && res.set) {
```

```
      res.set.call(this, this, val)
    }
  }
```

Vue.prototype.$set 方法首先通过 parseExpression 方法对 exp 表达式做解析，解析的结果包含 set 属性。set 属性是一个方法，它是通过调用 compileSetter(exp)方法生成的。compileSetter 的源码定义如下：

```
<!--源码目录：src/parsers/expression.js-->
function compileSetter (exp) {
  var path = parsePath(exp)
  if (path) {
    return function (scope, val) {
      setPath(scope, path, val)
    }
  } else {
    process.env.NODE_ENV !== 'production' && warn(
      'Invalid setter expression: ' + exp
    )
  }
}
```

compileSetter 方法返回了一个 function，前面执行 res.set.call(this, this, val)就相当于执行这里的 setPath(scope, path, val)方法。其中 scope 为 vm 实例；path 为 exp 解析后的路径数组，本例中为['b']；val 为设置的值，本例中为 2。接下来我们看一下 setPath 方法，它的源码定义如下：

```
<!--源码目录：src/parsers/path.js-->
export function setPath (obj, path, val) {
  var original = obj
  if (typeof path === 'string') {
    path = parse(path)
  }
  if (!path || !isObject(obj)) {
    return false
  }
  var last, key
  for (var i = 0, l = path.length; i < l; i++) {
    last = obj
    key = path[i]
    if (key.charAt(0) === '*') {
      key = parseExpression(key.slice(1)).get.call(original, original)
    }
```

```
    if (i < l - 1) {
      obj = obj[key]
      if (!isObject(obj)) {
        obj = {}
        if (process.env.NODE_ENV !== 'production' && last._isVue) {
          warnNonExistent(path, last)
        }
        set(last, key, obj)
      }
    } else {
      if (isArray(obj)) {
        obj.$set(key, val)
      } else if (key in obj) {
        obj[key] = val
      } else {
        if (process.env.NODE_ENV !== 'production' && obj._isVue) {
          warnNonExistent(path, obj)
        }
        set(obj, key, val)
      }
    }
  }
  return true
}
```

setPath 方法就是遍历 path 上的路径，对 path 路径上的对象求值，并调用 set(obj, key, val) 方法让 obj.key 也是响应式的。接下来我们看一下 set 方法，它的源码定义如下：

```
<!--源码目录：src/util/lang.js-->
export function set (obj, key, val) {
  if (hasOwn(obj, key)) {
    obj[key] = val
    return
  }
  if (obj._isVue) {
    set(obj._data, key, val)
    return
  }
  var ob = obj.__ob__
  if (!ob) {
```

```
    obj[key] = val
    return
  }
  ob.convert(key, val)
  ob.dep.notify()
  if (ob.vms) {
    var i = ob.vms.length
    while (i--) {
      var vm = ob.vms[i]
      vm._proxy(key)
      vm._digest()
    }
  }
  return val
}
```

本例中，我们调用 set 方法传入的 obj 是 Vue 实例，key 是 b，val 是 2，因此 obj._isVue 为 true，调用 set(obj._data, key, val)。在 Vue 实例化时，我们创建了 Observer 实例，并通过__ob__属性绑定在 data 上，因此可以拿到这个 ob 对象，并调用 ob.convert(key, val)方法把 key 绑定在 data 上，同时赋予 getter/setter 方法。然后调用 ob.dep.notify()通知订阅 Watcher 的更新，并最终更新视图。接下来判断 ob 上是否有 vue 实例，如果有则遍历 ob 上的所有 Vue 实例，调用 vm._proxy(key)方法把 key 代理到 vm 上，在本例中就是可以通过 vm.b 访问到 vm._data.b。最后调用 vm._digest()方法，强制 vm 上所有的 Watcher 重新计算。

经过这一系列的操作，就相当于给 Vue 实例的 data 对象新增了属性并让它也是响应式的。但在实际运用中，我们并不总是需要通过$set 方法新增属性，特别是在初始化数据时，接下来让我们看一下这部分内容。

20.3　初始化数据

尽管 Vue.js 提供了 API 动态地添加响应属性，但还是推荐 Vue 实例初始化时在 data 对象上声明所有的响应属性。

我们不建议这么做：

```
var vm = new Vue({
  template: '<div>{{msg}}</div>'
});
```

```
// 然后添加 `msg`
vm.$set('msg', DDFE!');
```

而是建议这么做：

```
var vm = new Vue({
  data: {
    // 以一个空值声明 `msg`
    msg: ''
  },
  template: '<div>{{msg}}</div>'
})
// 然后设置 `msg`
vm.msg = 'DDEF!'
```

之所以这么建议有两个原因：

- data 对象就像组件状态的模式（schema），在它上面声明所有的属性让开发者了解组件所期待的数据源的样子，这样的代码更易于理解。
- 添加一个顶级响应属性会调用 vm._digest()方法强制所有的 Watcher 重新计算，因为它之前不存在，没有 Watcher 追踪它。这么做性能通常是可以接受的（特别是对比 AngularJS 的脏检查），但是可以在初始化时避免。

20.4　异步更新队列

Vue.js 默认异步更新 DOM。正如我们前面介绍 Watcher 的实现那样，每当观察到数据变化时，Vue 就开始一个队列，将同一事件循环内所有的数据变化缓存起来。这样做的好处是，如果一个 Watcher 被多次触发，不会多次更新 DOM，只会推入一次到队列中，这样等到下一次事件循环时，Vue 只进行一次 DOM 更新，并清空队列。

内部异步队列的实现使用了 nextTick 方法，我们来了解一下它的实现原理。NextTick 方法的源码定义如下：

```
<!--源码目录：src/util/env.js-->
export const nextTick = (function () {
  var callbacks = []
  var pending = false
  var timerFunc
  function nextTickHandler () {
```

```
    pending = false
    var copies = callbacks.slice(0)
    callbacks = []
    for (var i = 0; i < copies.length; i++) {
      copies[i]()
    }
  }

  /* istanbul ignore if */
  if (typeof MutationObserver !== 'undefined' && !hasMutationObserverBug) {
    var counter = 1
    var observer = new MutationObserver(nextTickHandler)
    var textNode = document.createTextNode(counter)
    observer.observe(textNode, {
      characterData: true
    })
    timerFunc = function () {
      counter = (counter + 1) % 2
      textNode.data = counter
    }
  } else {
    // webpack attempts to inject a shim for setImmediate
    // if it is used as a global, so we have to work around that to
    // avoid bundling unnecessary code.
    const context = inBrowser
      ? window
      : typeof global !== 'undefined' ? global : {}
    timerFunc = context.setImmediate || setTimeout
  }
  return function (cb, ctx) {
    var func = ctx
      ? function () { cb.call(ctx) }
      : cb
    callbacks.push(func)
    if (pending) return
    pending = true
    timerFunc(nextTickHandler, 0)
  }
})()
```

nextTick 方法通过立即执行的匿名函数定义，在闭包环境下定义了 nextTickHandler 方法，作为最终执行回调函数的方法。nextTick 方法支持传入 cb 和 ctx 两个参数，其中 cb 代表异步执行的回调函数；ctx 代表回调函数执行的上下文环境。函数调用时，先用 callbacks 保存传入的回调函数，接着调用 timeFunc 异步执行 nextTickHandler 方法。

在这里 timeFunc 是根据当前浏览器的环境定义的，如果当前浏览器的环境支持 MutationObserver，则新建一个 MutationObserver 实例，传入 nextTickHandler 的回调函数，同时创建一个 TextNode DOM 对象，对它进行观察。MutationObserver 是监听 DOM 变动的接口，当 DOM 对象树发生任何变动时，MutationObserver 就会得到通知。当我们观察 TextNode 变化时，就会触发 MutationObserver 在实例化时传入的回调函数。因此，我们定义的 timerFunc 功能就是改变这个 TextNode 的 DOM，当 timeFunc 调用时就改变 TextNode 的值，从而触发了 nextTickHandler 的调用。如果当前浏览器不支持 MutationObserver，timerFunc 的定义就变成了 setImmediate 或者 setTimeout，通过这种方式达到异步执行 nextTickHandler 的目的。

在实际应用中，当我们通过数据驱动方式更新了某个 DOM，又想在 DOM 状态更新后做一些事情时，由于 DOM 异步更新的特性，不能在当前事件循环中直接操作这个 DOM，但可以通过 Vue 实例上的$nextTick 方法来实现。代码示例如下：

```
Vue.component('example', {
  template: '<span>{{msg}}</span>',
  data: function () {
    return {
      msg: 'DDFE'
    }
  },
  methods: {
    updateMessage: function () {
      this.msg = 'Hello DDFE'
      console.log(this.$el.textContent) // => 'DDFE'
      this.$nextTick(function () {
        console.log(this.$el.textContent) // => 'Hello DDFE'
      })
    }
  }
})
```

Vue 实例上的$nextTick 方法实际上就是调用了我们前面定义的 nextTick 方法，在这个方法的回调函数中，我们可以放心地操作更新后的 DOM 对象。

20.5　计算属性的奥秘

在 Vue.js 中，计算属性并不是简单的 getter，它会持续追踪它的响应依赖。在计算一个计算属性时，Vue.js 更新它的依赖列表并缓存结果，只有当其中一个依赖发生了变化时，缓存的结果才无效。因此，只要依赖不发生变化，访问计算属性就会直接返回缓存的结果，而不是调用 getter。

为什么要缓存？考虑这种场景：我们有一个高耗计算属性 A，它要遍历一个巨型数组并做大量的计算。然后，可能有其他的计算属性依赖 A。如果没有缓存，我们将调用 A 的 getter 许多次，超过必要的次数。

由于计算属性被缓存了，在访问它的时候 getter 不总是被调用。代码示例如下：

```
var vm = new Vue({
  el: '#main',
  data: {
    msg: 'hello'
  },
  computed: {
    example: function () {
      return Date.now() + this.msg
    }
  },
  created: function () {
    var me = this;
    setInterval(function () {
      console.log(me.example);
    }, 1000);
  }
});
```

计算属性 example 只有一个依赖：vm.msg。Date.now()不是响应依赖，因为它和 Vue 的数据观察系统无关。因此我们通过 setInterval 每隔 1s 访问一次 vm.example，得到的结果如图 20-7 所示。



图20-7　vm.example计算结果（1）

我们发现，每次访问 vm.example 的结果都不会改变，除非 vm.msg 改变。我们对代码稍微做一下改动：

```
created: function () {
   var me = this;
   setInterval(function () {
      me.msg = 'hello' + Math.random();
      console.log(me.example);
   }, 1000);
}
```

我们在计时器中修改了 vm.msg，这样每次访问 vm.example 的值都会改变，如图 20-8 所示。

```
1466754079162hello0.9337198741002137
1466754080160hello0.843598131798853
1466754081162hello0.9033000266602766
1466754082161hello0.0988812446371039
1466754083161hello0.018732661367093284
```

图20-8　vm.example计算结果（2）

有时候我们希望每次访问 vm.example 时都调用 getter，且不用修改它的响应依赖，这时可以为指定的计算属性关闭缓存。代码示例如下：

```
computed: {
   example: {
     cache: false,
     get: function () {
       return Date.now() + this.msg
     }
   }
},
created: function () {
   var me = this;
   setInterval(function () {
      console.log(me.example);
   }, 1000)
}
```

我们对指定的计算属性的 cahce 设置为 false，这样就不会缓存之前的计算结果了，每次都可以调用 vm.example 的 getter 方法，如图 20-9 所示。

```
1466767506989hello
1466767507989hello
1466767508990hello
1466767509989hello
1466767510987hello
1466767511988hello
```

图20-9　vm.example计算结果（3）

现在访问 vm.example，每次返回的时间戳都是新的。但是需要注意一点，只是在 JavaScript 中访问是这样的，数据仍然是依赖驱动的。如果在模板中绑定了{{example}}，只有响应依赖 vm.msg 变化才会更新 DOM。

接下来让我们一起从源码层面来探秘计算属性的实现。Vue 实例在创建过程中会有生命周期，其中有一个过程就是调用 vm._initComputed 处理 computed 属性。_initComputed 方法的源码定义如下：

```
<!--源码目录：src/instance/internal/state.js-->
Vue.prototype._initComputed = function () {
  var computed = this.$options.computed
  if (computed) {
    for (var key in computed) {
      var userDef = computed[key]
      var def = {
        enumerable: true,
        configurable: true
      }
      if (typeof userDef === 'function') {
        def.get = makeComputedGetter(userDef, this)
        def.set = noop
      } else {
        def.get = userDef.get
          ? userDef.cache !== false
            ? makeComputedGetter(userDef.get, this)
            : bind(userDef.get, this)
          : noop
        def.set = userDef.set
          ? bind(userDef.set, this)
          : noop
      }
```

```
        Object.defineProperty(this, key, def)
      }
    }
  }
```

Vue.prototype._initComputed 方法对 computed 属性进行遍历，拿到每一个计算属性的定义。计算属性的定义可以是一个 function，也可以是一个 object。默认计算属性是一个 function，只有 get 方法，如果想设置计算属性的 set 方法或者设置 cache 为 false，则应把计算属性定义成一个 object。计算属性的 get 方法默认是通过 makeComputedGetter 方法实现的，除非设置 cache 为 false。最后通过 Object.defineProperty(this, key, def)方法把每个计算属性绑定到 vm 实例上，访问 vm 上的计算属性就会调用 def.get 方法。接下来我们看一下 makeComputedGetter 方法的实现，源码定义如下：

```
<!--源码目录：src/instance/internal/state.js-->
function makeComputedGetter (getter, owner) {
    var watcher = new Watcher(owner, getter, null, {
      lazy: true
    })
    return function computedGetter () {
      if (watcher.dirty) {
        watcher.evaluate()
      }
      if (Dep.target) {
        watcher.depend()
      }
      return watcher.value
    }
}
```

makeComputedGetter 方法支持 getter 和 owner 两个参数。其中 getter 为用户定义的计算属性的 get 方法；owner 为当前 Vue 实例。首先创建一个 Watcher 实例，传入的 option 设置 lazy 属性值为 true，这个设置表明 Watcher 的求值被延迟了，并不会在创建时进行求值。接着返回 computedGetter 方法，访问 vm 上的计算属性就会调用该方法。当 watcher.dirty 为 true 时，会调用 watcher.evaluate 方法对 Watcher 进行计算。接下来我们看一下 watcher.evaluate 方法的实现，源码定义如下：

```
<!--源码目录：src/watcher.js-->
Watcher.prototype.evaluate = function () {
  // avoid overwriting another watcher that is being
```

```
  // collected.
  var current = Dep.target
  this.value = this.get()
  this.dirty = false
  Dep.target = current
}
```

Watcher.prototype.evaluate 方法通过 this.get 方法对 Watcher 进行求值，同时收集依赖。接着把 this.dirty 设置为 false，这样当再次访问 vm 上的计算属性时，watcher.dirty 为 false，就不会再次对 Watcher 求值了，因此也不会再次访问计算属性的 getter 方法了。

那么为何当我们修改计算属性的响应依赖时，getter 方法会再次被调用呢？因为在我们第一次调用计算属性的 getter 方法时，会访问响应依赖，也就触发了响应依赖的 getter 方法，把响应依赖添加到当前 Watcher 中，也把当前 Watcher 订阅到依赖的变化中。所以当响应依赖被修改时，就触发了响应依赖的 setter 方法，会调用 Watcher 的 update 方法。由于我们创建的是一个 lazy Watcher，所以会把 Watcher 的 dirty 属性值设置成 true。这样当我们再次访问计算属性时，会重新调用 watcher.evaluate 方法对 Watcher 求值，因此计算属性的 getter 方法被再次调用了。

20.6　总结

本章主要通过源码分析的方式带大家认识了 Vue.js 的响应式原理，了解到模型和视图是如何建立关联关系的、对一些 ES 5 检测不到的数据变化如何处理、如何初始化数据、Watcher 队列如何异步更新、计算属性如何实现等知识。

通过对 Vue.js 的一些内部实现细节的学习，有助于我们加深对 Vue.js 的理解，也会对我们使用 Vue 开发组件和项目有一定的指导意义，可以避免出现一些常见问题。

第 21 章

源码篇——父子类合并策略

相信在前面的组件篇中已经有同学看到了组件的 mixin 特性，里面也提到了两种合并策略。

当混合对象中出现下面两种情况时：

- 混合对象和组件存在同名的生命周期方法时，它们都会合并到一个数组中，混合对象的生命周期方法优先执行，组件的同名生命周期方法后执行。
- 混合对象的其他选项如 methods 中定义了和组件同名的方法时，组件会覆盖混合对象的同名方法。

21.1 策略是什么

策略其实是一个对象，在全局配置 config 中也暴露了一个对象，开发者可以自定义：

```
<!--源码目录：src/util/options.js -->
var strats = config.optionMergeStrategies = Object.create(null)
```

21.1.1 生命周期合并策略

当父子类同时存在同名的生命周期方法时，Vue.js 内部是如何处理的呢？生命周期合并策略如图 21-1 所示。

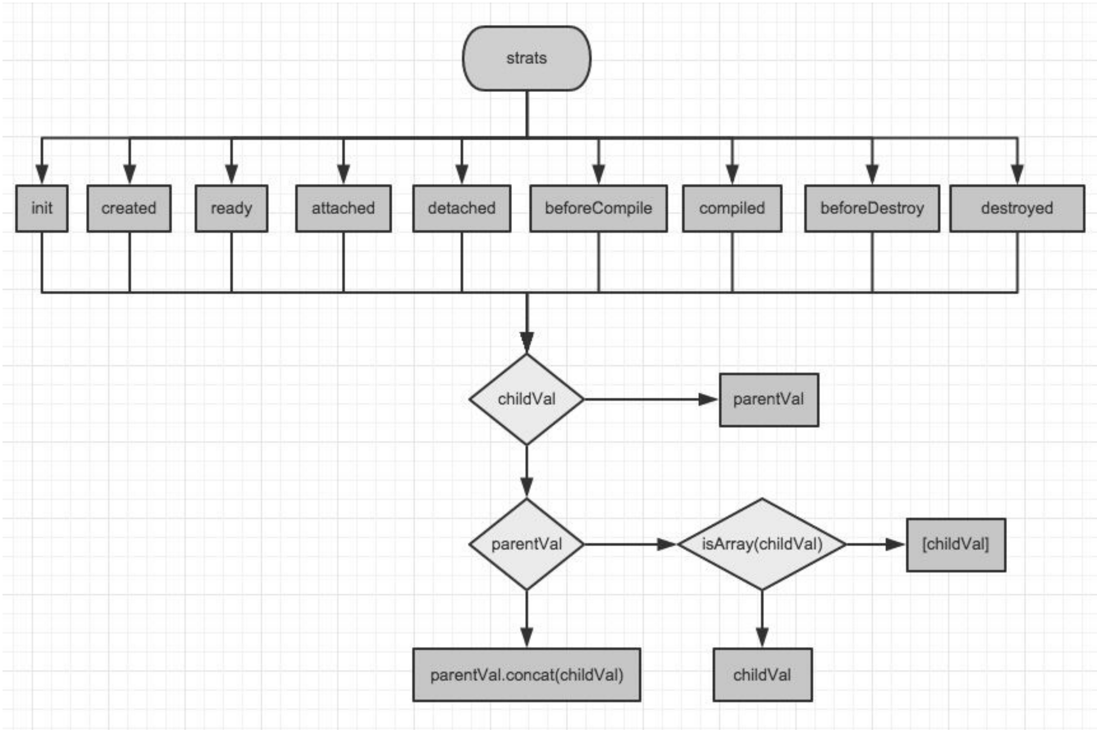


图21-1　生命周期合并策略

源码定义如下：

```
<!--源码目录：src/util/options.js -->
strats.init =
strats.created =
strats.ready =
strats.attached =
strats.detached =
strats.beforeCompile =
strats.compiled =
strats.beforeDestroy =
strats.destroyed = function (parentVal, childVal) {
  return childVal
    ? parentVal
      ? parentVal.concat(childVal)
      : isArray(childVal)
        ? childVal
```

```
        : [childVal]
    : parentVal
}
```

21.1.2　属性方法计算

相比前面的生命周期方法，我们来看看 props、methods 以及 computed 的合并策略，如图 21-2 所示。

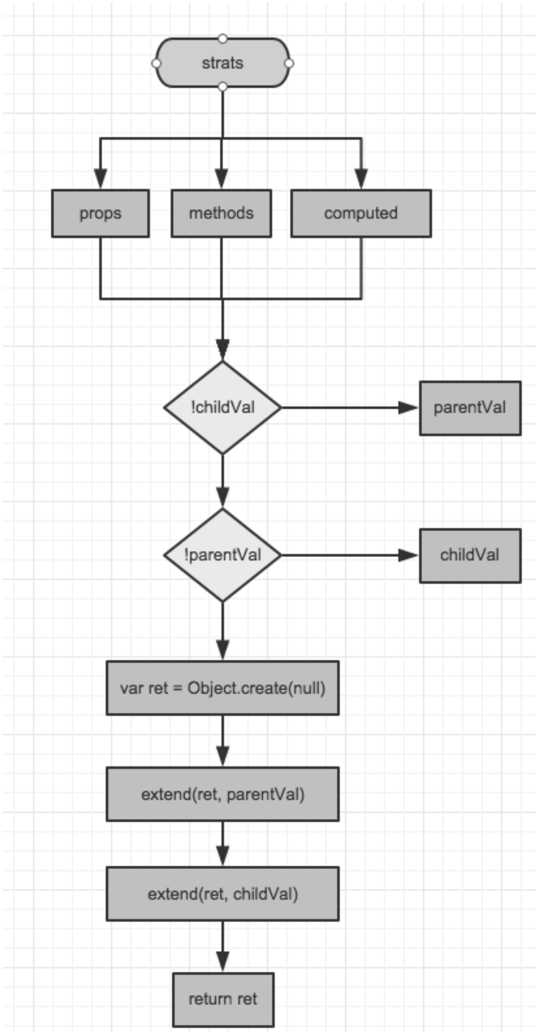


图21-2　属性方法计算的合并策略

源码定义如下：

```
<!--源码目录：src/util/options.js -->
strats.props =
strats.methods =
strats.computed = function (parentVal, childVal) {
  if (!childVal) return parentVal
  if (!parentVal) return childVal
  var ret = Object.create(null)
  extend(ret, parentVal)
  extend(ret, childVal)
  return ret
}
```

21.1.3 数据合并策略

相比前面两种模式，这种模式将 MVC 的 Controller 和 MVP 的 Presenter 改成了 ViewModel。View 的变化会自动更新到 ViewModel，同时 ViewModel 的变化也会自动同步到 View 上显示。

这种自动同步是因为 ViewModel 中的属性实现了 Observer，当属性变更时都能触发对应的操作，如图 21-3 所示。

源码定义如下：

```
<!--源码目录：src/util/options.js -->
strats.data = function (parentVal, childVal, vm) {
  if (!vm) {
    // in a Vue.extend merge, both should be functions
    if (!childVal) {
      return parentVal
    }
    if (typeof childVal !== 'function') {
      process.env.NODE_ENV !== 'production' && warn(
        'The "data" option should be a function ' +
        'that returns a per-instance value in component ' +
        'definitions.'
      )
```

```
      return parentVal
    }
    if (!parentVal) {
      return childVal
    }
    // when parentVal & childVal are both present,
    // we need to return a function that returns the
    // merged result of both functions... no need to
    // check if parentVal is a function here because
    // it has to be a function to pass previous merges.
    return function mergedDataFn () {
      return mergeData(
        childVal.call(this),
        parentVal.call(this)
      )
    }
  } else if (parentVal || childVal) {
    return function mergedInstanceDataFn () {
      // instance merge
      var instanceData = typeof childVal === 'function'
        ? childVal.call(vm)
        : childVal
      var defaultData = typeof parentVal === 'function'
        ? parentVal.call(vm)
        : undefined
      if (instanceData) {
        return mergeData(instanceData, defaultData)
      } else {
        return defaultData
      }
    }
  }
}
```

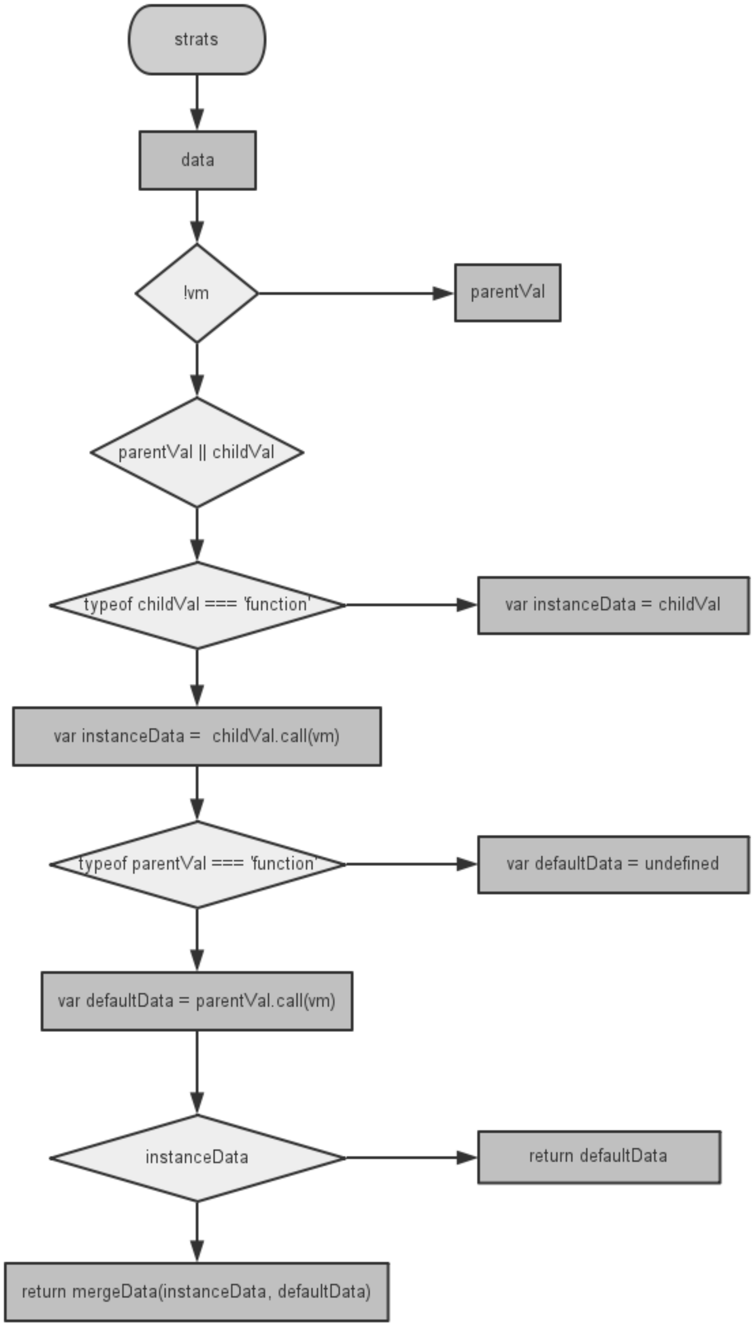


图21-3　数据的合并策略

第 22 章

源码篇——缓存

其实很多同学都看过 Vue.js 的源码，或者已经在前面的一些源码示例章节中看到了 Cache，本节我们就来讲述它。

比如在模板解析的源码中：

```
<!--源码目录：src/parsers/template.js -->
const templateCache = new Cache(1000)
// stringToFragment 方法中的应用
var hit = templateCache.get(cacheKey)
// stringToFragment 方法中的应用
templateCache.put(cacheKey, frag)
```

22.1　Cache 有什么用

顾名思义，缓存一般可以用来放置一些数据，同时提供一些 API 来操作数据，如图 22-1 所示。

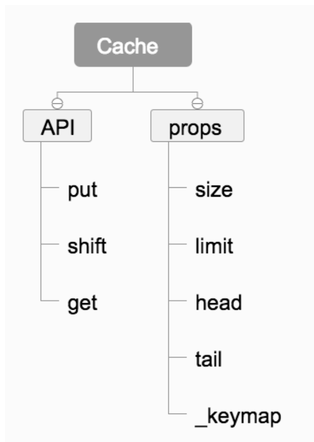


图22-1　Cache对象

Vue 作者在 Vue 源码中提到 Cache 参考了 rsms 的 js-lru，原文如下：

```
A doubly linked list-based Least Recently Used (LRU) cache.
Will keep most recently used items while discarding least recently used items when its
limit is reached.
This is a bare-bone version of Rasmus Andersson's js-lru
```

一个基于 LRU 算法的 Cache，Rasmus Andersson 写的 js-lru 的精简版本实现。js-lru 本身提供了更多的 API，比如 find、set、remove、removeAll 等。

22.2　LRU

LRU（Least Recently Used，最近最少使用），有计算机学习背景的同学可能相对比较熟悉，类似操作系统里面的内存管理，如图 22-2 所示。

```
Illustration of the design:

      entry             entry             entry             entry
      ______            ______            ______            ______
     | head |.newer => |      |.newer => |      |.newer => | tail |
     |  A   |          |  B   |          |  C   |          |  D   |
     |______| <= older.|______| <= older.|______| <= older.|______|

  removed  <--  <--  <--  <--  <--  <--  <--  <--  <--  <--  <--  added
```

图22-2　js-lru的设计图

22.3　Cache 类

Cache 类包含一些基本属性和几个原型链方法，接受一个参数 limit。

```
<!--源码目录：src/cache.js#14 -->
export default function Cache (limit) {
  this.size = 0
  this.limit = limit
  this.head = this.tail = undefined
  this._keymap = Object.create(null)
}
```

22.4　put

put 接受两个参数：一个 key 和一个 value，返回被删除的。

```
<!--源码目录：src/cache.js#34 -->
p.put = function (key, value) {
  var removed
  if (this.size === this.limit) {
    removed = this.shift()
  }
```

```
  var entry = this.get(key, true)
  if (!entry) {
    entry = {
      key: key
    }
    this._keymap[key] = entry
    if (this.tail) {
      this.tail.newer = entry
      entry.older = this.tail
    } else {
      this.head = entry
    }
    this.tail = entry
    this.size++
  }
  entry.value = value

  return removed
}
```

22.5 shift

shift 删除 Cache 里面最近最少使用的项，返回被删除的。

```
<!--源码目录：src/cache.js#66 -->
p.shift = function () {
  var entry = this.head
  if (entry) {
    this.head = this.head.newer
    this.head.older = undefined
    entry.newer = entry.older = undefined
    this._keymap[entry.key] = undefined
    this.size--
  }
  return entry
}
```

22.6 get

get 接受参数 key，返回对应的值。

```
<!--源码目录：src/cache.js#87 -->
p.get = function (key, returnEntry) {
  var entry = this._keymap[key]
  if (entry === undefined) return
  if (entry === this.tail) {
    return returnEntry
      ? entry
      : entry.value
  }
  // HEAD--------------TAIL
  //   <.older   .newer>
  //  <--- add direction --
  //   A  B  C  <D>  E
  if (entry.newer) {
    if (entry === this.head) {
      this.head = entry.newer
    }
    entry.newer.older = entry.older // C <-- E.
  }
  if (entry.older) {
    entry.older.newer = entry.newer // C. --> E
  }
  entry.newer = undefined // D --x
  entry.older = this.tail // D. --> E
  if (this.tail) {
    this.tail.newer = entry // E. <-- D
  }
  this.tail = entry
  return returnEntry
    ? entry
    : entry.value
}
```

第 23 章

源码篇——属性 props

其实很多同学都使用过 props 配置给组件元素设置一些属性，本章我们从源码角度来分析一下相关的具体实现。

23.1 流程设计

我们看一下初始化 props 的流程，如图 23-1 所示。

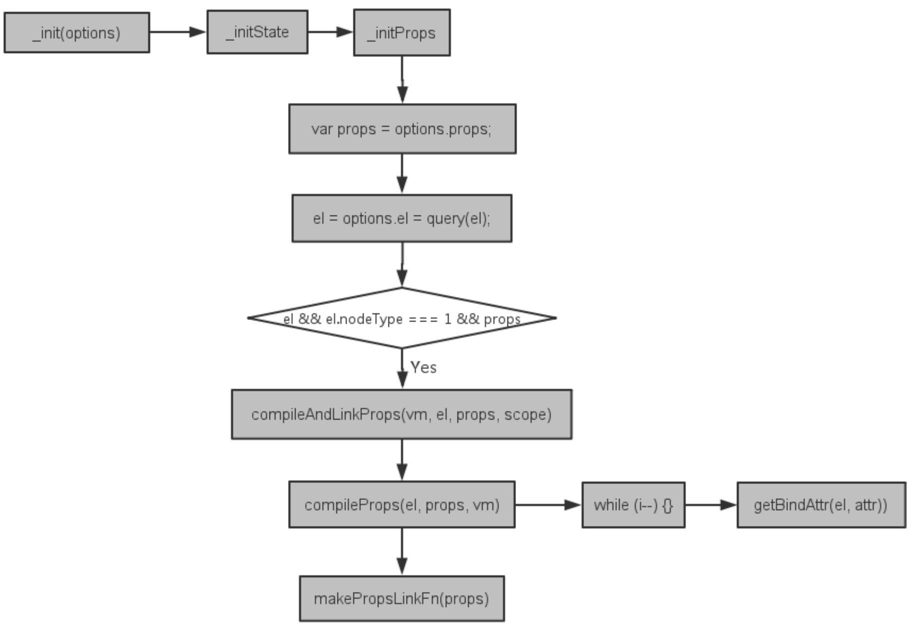


图23-1 props初始化流程图

我们设计了一个 didi-list 列表组件，它有两个属性：

```
<!-- didi-list props -->
export default {
  props: {
    gridData: Array,
    fields: Array
  }
}
```

compileProps 解析后的结果如图 23-2 所示。

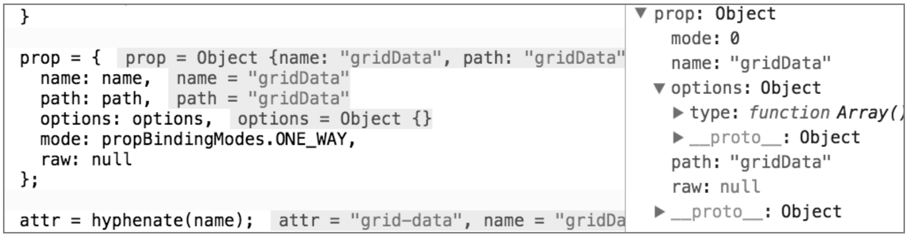


图23-2 compileProps转换gridData

gridData 会通过 hyphenate 函数将 gridData 转换成符合 kebab-case（短横线）规则的 grid-data，如图 23-3 所示。

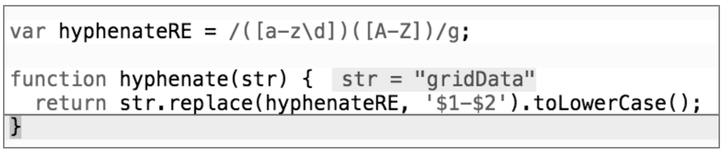


图23-3 hyphenate实现转换

如图 23-4 所示是 filelds 的转换。

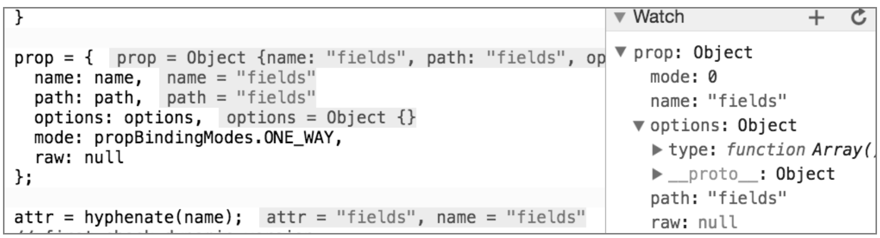


图23-4 compileProps转换fields

如图 23-5 所示是 gridData 的转换成内部 prop 对象。

如图 23-6 所示是 fields 的转换成内部 prop 对象。

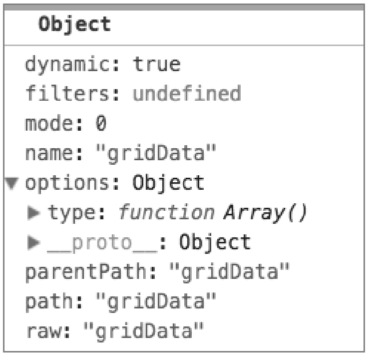


图23-5　gridData的对象prop

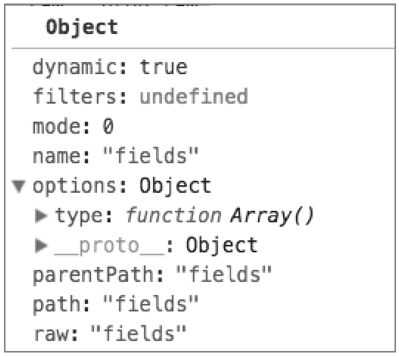


图23-6　fields的对象prop

23.2　属性 name

其实在编译 props 时，会对属性 name 进行一些验证处理。

看下面的源码片段：

```
<!--源码目录：src/compiler/compile-props.js -->
export function compileProps (el, propOptions, vm) {
  var props = []
  var names = Object.keys(propOptions)
  var i = names.length
  var options, name, attr, value, path, parsed, prop
  while (i--) {
    name = names[i]
    options = propOptions[name] || empty

    if (process.env.NODE_ENV !== 'production' && name === '$data') {
      warn('Do not use $data as prop.', vm)
      continue
    }

    // props could contain dashes, which will be
    // interpreted as minus calculations by the parser
    // so we need to camelize the path here
    path = camelize(name)
```

```
    if (!identRE.test(path)) {
      process.env.NODE_ENV !== 'production' && warn(
        'Invalid prop key: "' + name + '". Prop keys ' +
        'must be valid identifiers.',
        vm
      )
      continue
    }
}
```

23.3 coerce

在 coerce 属性配置上我们可以设置转换函数。

看下面的源码片段：

```
<!--源码目录：src/compiler/compile-props.js -->
function coerceProp (prop, value) {
  var coerce = prop.options.coerce
  if (!coerce) {
    return value
  }
  // coerce is a function
  return coerce(value)
}
```

23.4 type 验证

type 可以设置常见的类型，也支持自定义。

```
<!--源码目录：src/compiler/compile-props.js -->
function assertType (value, type) {
  var valid
  var expectedType
  if (type === String) {
    expectedType = 'string'
    valid = typeof value === expectedType
  } else if (type === Number) {
    expectedType = 'number'
```

```
    valid = typeof value === expectedType
  } else if (type === Boolean) {
    expectedType = 'boolean'
    valid = typeof value === expectedType
  } else if (type === Function) {
    expectedType = 'function'
    valid = typeof value === expectedType
  } else if (type === Object) {
    expectedType = 'object'
    valid = isPlainObject(value)
  } else if (type === Array) {
    expectedType = 'array'
    valid = isArray(value)
  } else {
    valid = value instanceof type
  }
  return {
    valid,
    expectedType
  }
}
```

23.5　default

default 可以设置一些默认值。

看下面的源码片段：

```
<!--源码目录：src/compiler/compile-props.js -->
function getPropDefaultValue (vm, prop) {
  // no default, return undefined
  const options = prop.options
  if (!hasOwn(options, 'default')) {
    // absent boolean value defaults to false
    return options.type === Boolean ? false : undefined
  }
  var def = options.default
  // warn against non-factory defaults for Object & Array
  if (isObject(def)) {
```

```
    process.env.NODE_ENV !== 'production' && warn(
      'Invalid default value for prop "' + prop.name + '": ' +
      'Props with type Object/Array must use a factory function ' +
      'to return the default value.',
      vm
    )
  }
  // call factory function for non-Function types
  return typeof def === 'function' && options.type !== Function
    ? def.call(vm)
    : def
}
```

23.6　validator

validator 可以设置自定义的验证函数。

看下面的源码片段：

```
<!--源码目录：src/compiler/compile-props.js -->
var validator = options.validator
  if (validator) {
    if (!validator(value)) {
      process.env.NODE_ENV !== 'production' && warn(
        'Invalid prop: custom validator check failed for prop "' + prop.name + '".',
        vm
      )
      return false
    }
}
```

第 24 章

源码篇——events

前面我们介绍过通过 methods 对象配置来给模板 DOM 元素绑定事件，那么如何在 Vue.js 实例之间以及父子类之间通过事件来通信呢？我们可以通过 events 这个配置项来实现。

24.1　events 配置是什么

其实可以理解为一个简单的配置对象：

- Key 是在实例事件比如$emit 调用时传入的参数。
- Value 是处理函数（当然也可以是 methods 里面配置的方法名）。我们可以在 Vue 实例化时通过类似 methods 的配置项 events。

24.2　如何配置

和 methods 配置一样，也是一个对象。Key 是监听的事件名称，Value 是对应的处理函数。在 new Vue 的实例化过程中，Vue 实例会调用 events 对象的每一个 Key，如图 24-1 所示。

图 24-1 比较大，我们可以逐步拆解，看如下代码示例：

```
<script>
<!-- new Vue 实例化 -->
var vm = new Vue({
    events: {
        sayHi: function (msg) {
            console.log("Hello, this is:" + msg);
        }
    }
})
</script>
```

我们在实例化时传入了一个 events 对象，如图 24-2 所示。



图24-1　events初始化流程图

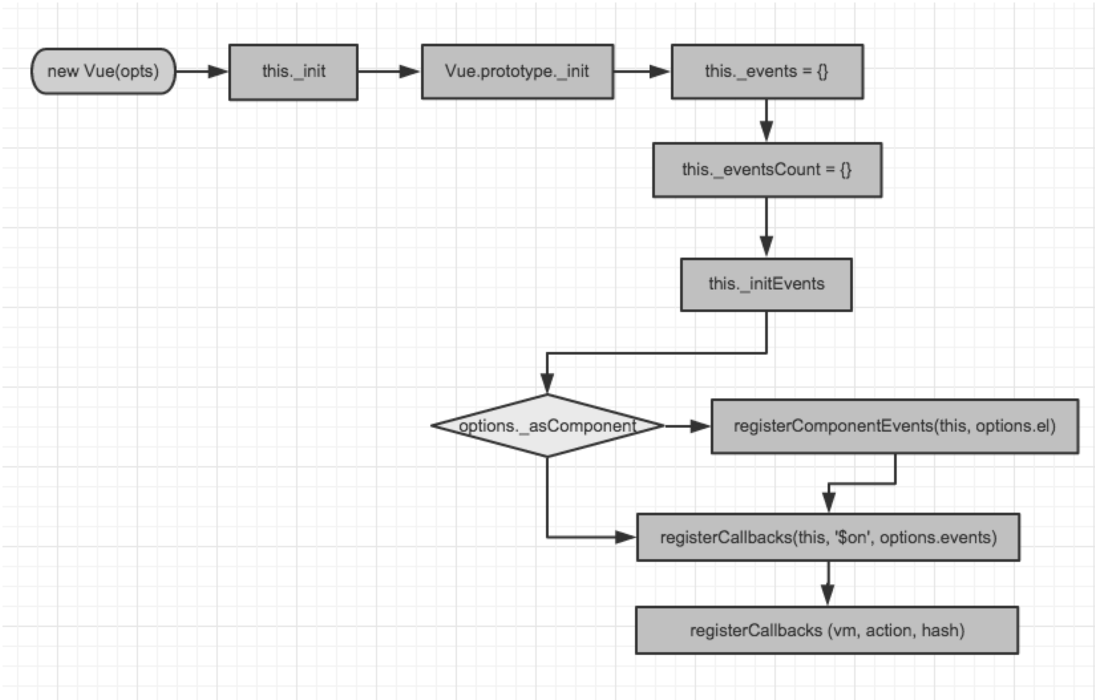


图24-2 实例化methods第一步

registerCallbacks 和 register 执行流程分别如图 24-3、图 24-4 所示。

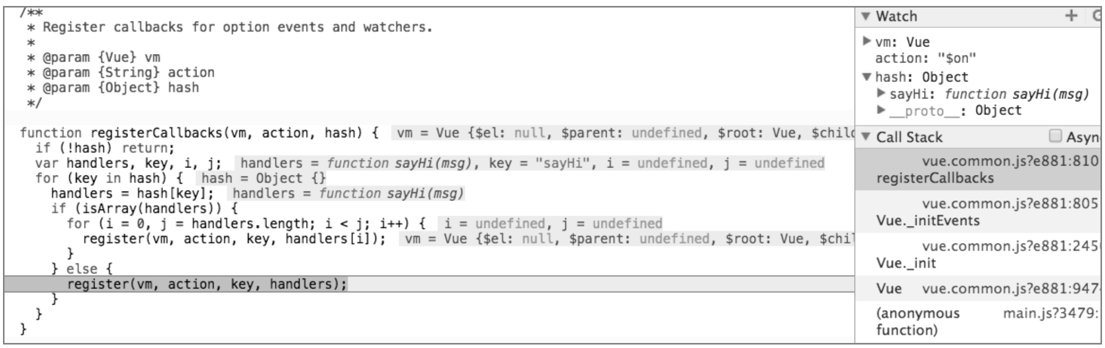


图24-3 registerCallbacks执行流程图

events 的$on 执行流程如图 24-5 所示。它操作 this._events 对象，把我们之前传入的 sayHi 当成 key，对应的处理函数当成它的处理函数。

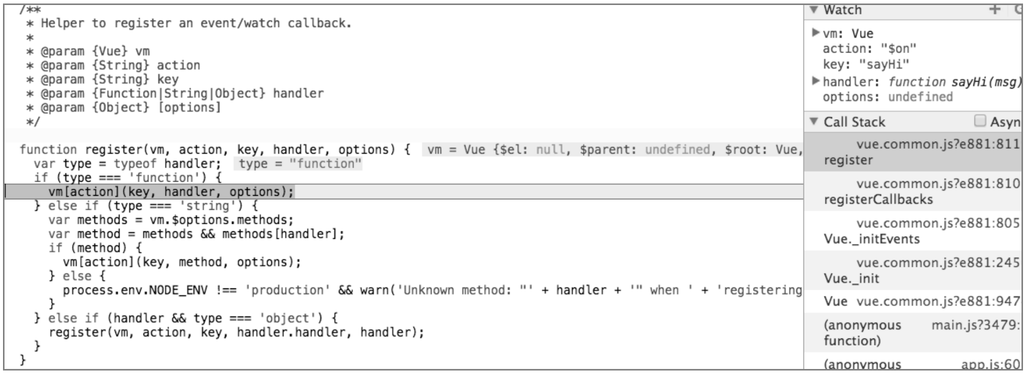


图24-4　register执行流程图

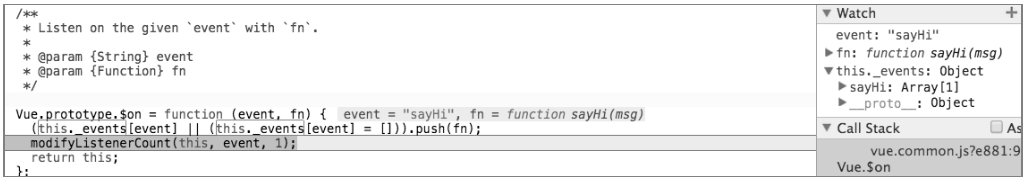


图24-5　events的$on执行流程图

events 的 modifyListenerCount 执行流程如图 24-6 所示。

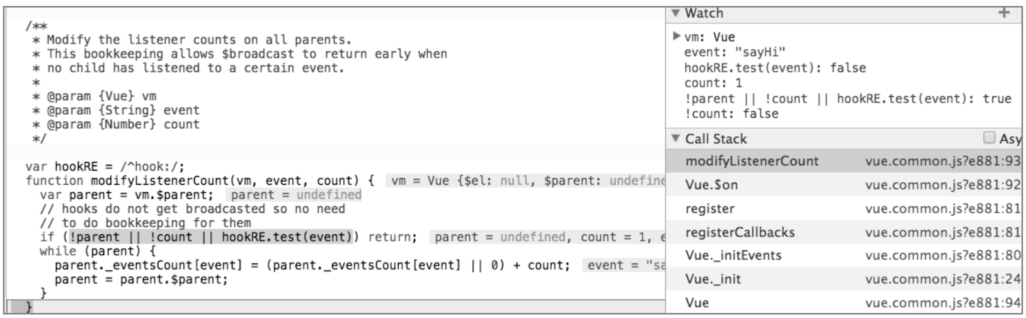


图24-6　events的modifyListenerCount执行流程图

24.2.1　$emit 触发

了解了 events 参数中事件是如何绑定的，现在我们来看一下如何触发它，使用$emit，语法如下：

```
vm.$emit(event,[…args])
```

```
<script>
<!-- new Vue 实例化 -->
var vm = new Vue({
  events: {
    sayHi: function (msg) {
      console.log('Hello, this is:' + msg)
    }
  }
})

<!-- $emit 触发-->
vm.$emit('sayHi', 'DDFE')
</script>
```

vm 调用$emit 的调试如图 24-7 所示。

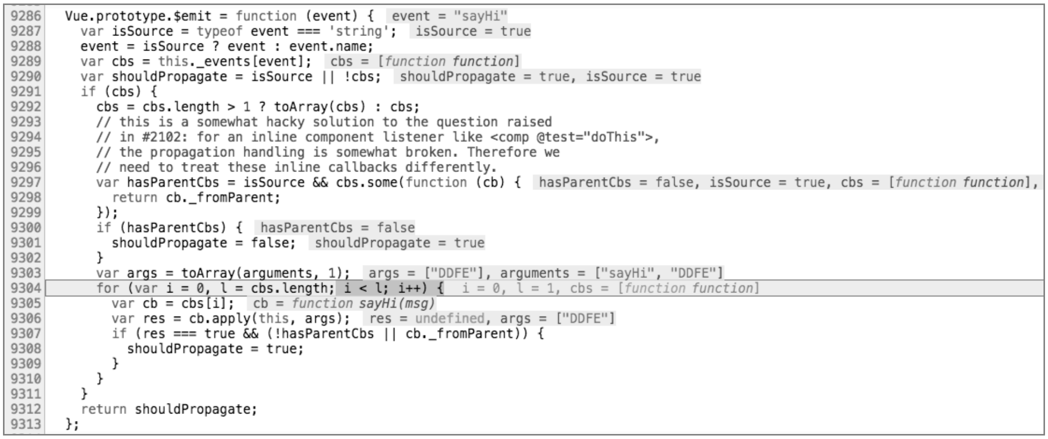


图24-7　$emit调试

我们来看如下源码实现：

```
<!--源码目录：src/instance/api/events.js#91 -->
Vue.prototype.$emit = function (event) {
  // 我们传入的是字符串类型的 sayHi
  // 所以 isSource 返回 true
  var isSource = typeof event === 'string'
  event = isSource ? event : event.name
  // 直接从 this._events 这个对象里面获取
  var cbs = this._events[event]
  // 这里返回 true
```

```
  var shouldPropagate = isSource || !cbs
  if (cbs) {
      cbs = cbs.length > 1 ? toArray(cbs) : cbs
      var hasParentCbs = isSource && cbs.some(function (cb) {
        return cb._fromParent
      })
      if (hasParentCbs) {
        shouldPropagate = false
      }
      var args = toArray(arguments, 1)
      for (var i = 0, l = cbs.length; i < l; i++) {
        var cb = cbs[i]
        // 最终还是调用 apply 把上面传递的参数再传给监听函数
        var res = cb.apply(this, args)
        if (res === true && (!hasParentCbs || cb._fromParent)) {
          shouldPropagate = true
        }
      }
    }
    return shouldPropagate
}
```

24.2.2　$once 绑定

相比前面流程示例中的$on，$once 绑定的事件只触发一次，触发完成后就删除了。

$on 的语法如下：

```
vm.$once(event,callback)
```

我们来看如下源码实现：

```
<!--源码目录：src/instance/api/events.js#26 -->
Vue.prototype.$once = function (event, fn) {
  var self = this
  function on () {
    // 依赖 $off
    self.$off(event, on)
    // 最终还是 apply
    fn.apply(this, arguments)
  }
```

```
  on.fn = fn
  // 依赖 $on
  this.$on(event, on)
  return this
}
```

24.2.3　$off 删除

顾名思义，$off 删除事件监听函数。

$off 的语法如下：

```
vm.$off([event,callback])
```

- 没有参数，删除所有的事件监听函数。
- 只有事件名称，删除这个事件名称对应的所有监听函数。
- 同时指定事件名称和对应的监听函数，只删除对应的。

我们来看如下源码实现：

```
<!--源码目录：src/instance/api/events.js#45 -->
Vue.prototype.$off = function (event, fn) {
  var cbs
  // all
  // !0 -> true
  if (!arguments.length) {
    if (this.$parent) {
      for (event in this._events) {
        cbs = this._events[event]
        if (cbs) {
          modifyListenerCount(this, event, -cbs.length)
        }
      }
    }
    // 重置所有
    this._events = {}
    return this
  }
  // specific event
  cbs = this._events[event]
  if (!cbs) {
    return this
```

```
  }
  if (arguments.length === 1) {
    // 删除这个事件对应的所有监听函数
    modifyListenerCount(this, event, -cbs.length)
    this._events[event] = null
    return this
  }
  // specific handler
  var cb
  var i = cbs.length
  while (i--) {
    cb = cbs[i]
    if (cb === fn || cb.fn === fn) {
      modifyListenerCount(this, event, -1)
      cbs.splice(i, 1)
      break
    }
  }
  return this
}
```

24.2.4　$dispatch 派发

首先在实例上触发$on 对应绑定的监听函数，然后沿着$parent 向上冒泡。如果返回 false，就直接 return 了。

$dispatch 的语法如下：

```
vm.$dispatch(event,[…args])
```

我们来看如下源码实现：

```
<!--源码目录：src/instance/api/events.js#163 -->
Vue.prototype.$dispatch = function (event) {
  // 先调用$emit 执行一次
  var shouldPropagate = this.$emit.apply(this, arguments)
  // 判断参数，false 就 return 了
  if (!shouldPropagate) return
  var parent = this.$parent
  var args = toArray(arguments)
  // use object event to indicate non-source emit
  // on parents
  args[0] = { name: event, source: this }
  // 沿着父链
```

```
  while (parent) {
    shouldPropagate = parent.$emit.apply(parent, args)
    parent = shouldPropagate
      ? parent.$parent
      : null
  }
  return this
}
```

24.2.5　$broadcast 广播

遍历当前所有$children，需要返回 true，不然就中止了。

$broadcast 的语法如下：

```
vm.$broadcast(event,[…args])
```

我们来看如下源码实现：

```
<!--源码目录：src/instance/api/events.js#131 -->
Vue.prototype.$broadcast = function (event) {
  var isSource = typeof event === 'string'
  event = isSource? event : event.name
  // if no child has registered for this event,
  // then there's no need to broadcast.
  if (!this._eventsCount[event]) return
  var children = this.$children
  var args = toArray(arguments)
  if (isSource) {
    // use object event to indicate non-source emit
    // on children
    args[0] = { name: event, source: this }
  }
  for (var i = 0, l = children.length; i < l; i++) {
    var child = children[i]
    var shouldPropagate = child.$emit.apply(child, args)
    // 子类的子类必须返回 true
    if (shouldPropagate) {
      child.$broadcast.apply(child, args)
    }
  }
  return this
}
```

第 25 章 Webpack

Webpack 是一个模块化加载器，它同时支持 AMD、CMD 等加载规范。与其他模块化加载器相比，它具有以下优势：

1. 代码分割

Webpack 支持两种依赖加载：同步和异步。同步的依赖会在编译时直接打包输出到目的文件中；异步的依赖会单独生成一个代码块，只有在浏览器中运行需要的时候才会异步加载该代码块。

2. Loaders

在默认情况下，Webpack 只能处理 JS 文件，但是通过加载器我们可以将其他类型的资源转换为 JS 输出。

3. 插件机制

Webpack 提供了强大的插件系统，当 Webpack 内置的功能不能满足我们的构建需求时，我们可以通过使用插件来提高工作效率。

25.1 安装

全局安装，执行如下命令：

```
$ npm i webpack -g
```

除了全局安装，我们也可以将 Webpack 作为项目依赖在项目中进行安装。这样做的好处是，我们可以在不同的项目中使用不同的 Webpack 版本。

首先，我们使用 npm 命令初始化 npm 项目，执行如下命令：

```
$ npm init
```

然后，运行以下命令在项目中安装 Webpack，并将其写入 package.json 依赖字段 devDependencies 中：

```
$ npm i webpack -save-dev
```

25.2　基本使用

我们先来看一个最简单的基于 Webpack 命令行参数打包的例子。假定有以下目录结构：

```
example
  |- app.js
  |- cats.js
```

下面我们看看各文件内容，基于 CommonJS 规范引用依赖文件，代码示例如下：

```
// cats.js
var cats = ['dave', 'henry', 'martha']
module.exports = cats
// app.js 入口文件
cats = require('./cats.js')
console.log(cats)
```

app.js 是项目的 JavaScript 入口文件，Webpack 将会从该文件开始对依赖文件进行打包。

我们通过 Webpack 命令指定要打包的入口文件 app.js 和最终输出文件 app.bundle.js 来打包应用，执行如下命令：

```
$ webpack ./app.js app.bundle.js
```

以上命令运行后，Webpack 会解析依赖的文件，然后打包输出到 app.bundle.js 文件。图 25-1 演示了 Webpack 打包过程。

现在我们可以在 node 中运行打包后的 app.bundle.js 文件来看看效果，执行如下命令：

```
$ node app.bundle.js
["dave", "henry", "martha"]
```

我们在 node 环境中进行了演示，实际上 Webpack 打包后的代码可以运行在任何环境中，包括浏览器环境。

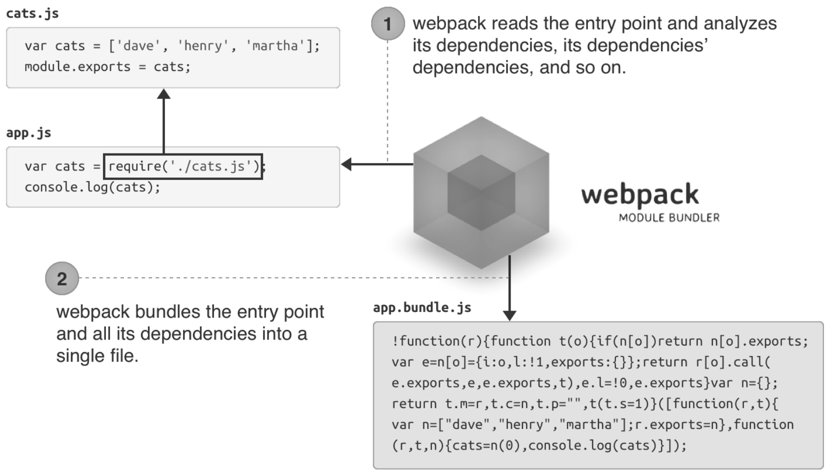


图25-1　Webpack打包过程

25.3　命令行

在 25.2 节中，我们学习了如何使用 Webpack 命令进行简单打包，语法如下：

```
$ webpack <entry><output>
```

entry 为要打包的入口文件路径，output 为打包后的文件路径。

Webpack 命令还提供了很多参数供我们自定义打包过程，下面介绍一些常用参数。

- -p，对打包后的代码进行压缩。
- --watch，文件发生变化时，重新打包。
- --config，指定 Webpack 打包配置文件，稍后会详细介绍配置文件。
- --progress，在终端显示打包过程。

25.4　配置文件

通过 Webpack 命令行传参，我们可以进行简单的打包构建。对于复杂的打包，我们可以在项目的根目录下提供一个配置文件，在配置文件中对打包过程进行更详细的配置。在项目根目录下不提供参数直接调用 webpack 命令：

```
$ webpack
```

Webpack 默认会调用项目根目录下的 webpack.config.js 文件，我们也可以通过-config 参数指定配置文件，执行如下命令：

```
$ webpack -config webpack.config.build.js
```

配置文件的内容需要通过 module.exports 进行导出，代码示例如下：

```
// webpack.config.js
module.exports = {
  // 配置选项
}
```

现在我们看一下 Webpack 中包含的配置选项，如图 25-2 所示。

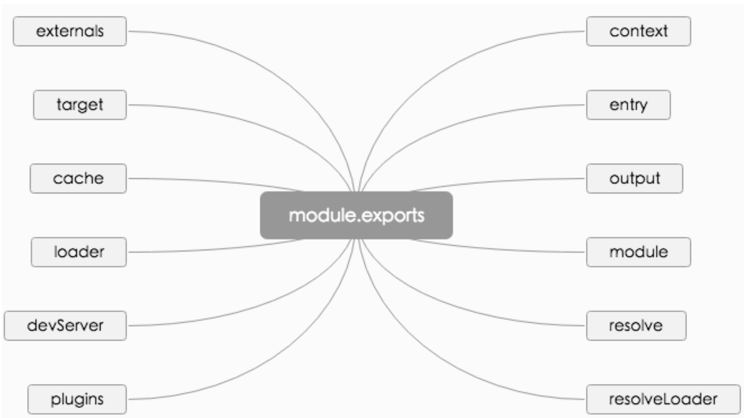


图25-2　配置选项

接下来介绍几个常用选项的含义及用法。

25.4.1　context

context 选项用来配置基础路径（必须为绝对路径），默认为 process.cwd()，即运行 webpack 命令的目录。

25.4.2　entry

entry 选项用来配置要打包的入口文件，值可以是字符串、数组、对象。该选项指定的路径会相对 context 选项指定的路径进行查找。

1. 字符串

直接指定路径，该路径相对于 context 选项。

```
entry: "./entry"
```

2. 数组

路径数组，Webpack 会按序打包，但是只导出最后一个文件。

```
entry: ["./entry1", "./entry2"]
```

3. 对象

当 entry 值为对象时，键名为块名，可以随意指定，键值可以为字符串或数组类型。该块名可以在 output 选项中使用，代码示例如下：

```
{
  entry: {
    page1: "./page1",
    page2: ["./entry1", "./entry2"]
  },
  output: {
    // 打包后输出文件名，name 为 entry 中对应的键名
    filename: "[name].bundle.js"
  }
}
```

以上选项配置后，运行命令在项目根目录下会生成 page1.bundle.js 和 page2.bundle.js 文件。

25.4.3 output

output 选项可用来配置输出信息：

```
output.filename
```

配置打包后的文件名，注意值不是绝对路径。我们应该通过 output.path 来指定输出路径，filename 会相对 output.path 来输出，代码示例如下：

```
// 单入口示例
{
  entry: './src/app.js',
  output: {
    filename: 'bundle.js',
```

```
    path: __dirname + '/build'
  }
}
// 写入磁盘路径为./build/bundle.js
```

如果项目有多个入口，对于每个入口打包后的文件名我们需要保证其唯一性。Webpack 提供了以下模式来动态生成输出文件名：

- [name]，入口文件块名。
- [hash]，每个入口打包后的 hash 值。
- [chunkhash]，在使用代码分割时，异步加载的文件的 hash 值。

```
// 多入口示例
{
  entry: {
    app: './src/app.js',
    search: './src/search.js'
  },
  output: {
    filename: '[name].js',
    // filename: '[hash].js',
    // filename: '[chunkhash].js',
    path: __dirname + '/build'
  }
}
// 写入磁盘路径为./build/app.js、./build/search.js
```

output.path——打包后的文件的根目录（绝对路径）。

25.4.4　module

module 选项用来进行模块加载相关配置。

module.loaders——加载器数组，当依赖文件匹配指定的 test 模式时，Webpack 会自动调用数组中的相应加载器去处理该文件，然后返回 JS 格式的文件。

加载器是一个对象，该对象拥有以下属性：

- test —— 正则表达式，Webpack 用其去匹配相应的文件，通常用来匹配文件后缀。
- exclude —— 不应该被 loader 处理的文件。

- include —— 一个路径数组，这些路径将会被 loader 处理。
- loader —— test 匹配到的文件对应的加载器，值是一个加载器名字字符串，多个加载器之间用“!”分隔。

代码示例如下：

```
module: {
  loaders: [
    {
     // 匹配 jsx 后缀的文件
     test: /\.jsx$/,
     // include 中的目录会被 loader 解析
     include: [
       path.resolve(__dirname, "app/src"),
       path.resolve(__dirname, "app/test")
     ],
     // babel loader，该 loader 可以用来解析 ES 6 语法
     loader: "babel-loader" // 或者 "babel", Webpack 将会自动添加 '-loader'
    }
  ]
}
```

25.4.5 resolve

resolve 选项用来配置依赖文件的匹配，如依赖文件别名配置、模块的查找目录、默认查找的文件后缀等。

1. resolve.alias

该选项用来配置依赖文件的别名，值是一个对象，该对象的键是别名，值是实际路径。

2. resolve.root

该选项用来指定模块的查找根路径，必须为绝对路径，值可以是路径字符串或者路径数组。若是路径数组，Webpack 会依次在这些路径中查找，如果找到则终止；否则会继续在下一个路径中查找。代码示例如下：

```
// webpack.config.js
var path = require('path');
// ...
```

```
resolve: {
  root: [
    path.resolve('./app/modules'),
    path.resolve('./vendor/modules')
  ]
}
```

3. resolve.modulesDirectories

该选项用来指定模块目录，值是一个路径数组，默认值为["web_modules", "node_modules"]。

25.4.6　devServer

devServer 选项可用来配置 webpack-dev-server 的行为。以下代码用来指定服务的根路径：

```
{
  devServer: {
    contentBase: "./build",
  }
}
```

25.5　开发调试

开发代码时，调试是必不可少的。我们可以使用 webpack-dev-server 在浏览器中进行调试。

webpack-dev-server 是一个基于 Express 的 Node.js 服务器。在文件发生改变时，它会自动触发打包过程，然后通过 Socket.IO 通知浏览器刷新页面，可以大大提高工作效率。

25.5.1　安装

```
$ npm i -g webpack-dev-server
```

25.5.2　启动服务

我们可以运行以下命令来启动服务：

```
$ webpack-dev-server
```

不带参数运行以上命令，默认会读取 webpack.config.js 进行打包，我们可以通过-config 来指定配置文件。详情请参阅 25.5.3 节。

25.5.3　命令行参数

所有的 Webpack 命令接受的参数，webpack-dev-server 都可以接受。除此之外，我们还可以向 webpack-dev-server 传递额外的参数。下面我们来看一些常用的参数。

- --content-base，指定请求的根路径。
- --host，指定服务端监听的地址可以是 IP 地址或者域名。当值为 0.0.0.0 时，可以监听一台机器的所有 IP 地址，如 127.0.0.1 或机器在局域网中的 IP 地址。
- --port，指定服务端监听的端口号。
- --compress，启用 gzip 压缩。
- --inline，自动将 Socket.IO 代码注入到打包后的文件中。启用该选项，当文件内容改变时可以自动刷新浏览器。

25.5.4　配置文件

除了通过命令行传参来配置 webpack-dev-server 外，我们还可以通过 Webpack 配置文件如默认的 webpack.config.js 中的 devServer 选项对其进行配置，所有命令行参数都支持在配置文件中进行设定。例如：

```
// webpack.config.js
module.exports = {
  // ...
  devServer: {
    inline: true
  }
}
```

25.6　使用插件

在 Webpack 提供的基本功能不能满足需求的情况下，Webpack 还允许我们使用插件来控制打包的各个过程。

25.6.1　安装

我们需要通过 npm 安装相关的插件，这里以 WebpackBrowserPlugin 为例，该插件用来在 Webpack 或 webpack-dev-server 运行完成后启动浏览器。

首先安装插件，执行如下命令：

```
$ npm install --save-dev webpack-browser-plugin
```

然后在 webpack.config.js 中引用插件，并在插件选项中注册该插件：

```
// webpack.config.js
var WebpackBrowserPlugin = require('webpack-browser-plugin');
module.exports = {
  ...
  ...
  plugins: [
    new WebpackBrowserPlugin()
  ],
  ...
}
```

如果是 Webpack 内置的插件，首先需要在项目中安装 Webpack，执行如下命令：

```
$ npm install -save-dev webpack
```

然后在配置文件中引用并注册插件。这里以 DefinePlugin 为例，该插件可以在打包时替换指定变量：

```
// webpack.config.js
// 首先需要引入 Webpack
var webpack = require('webpack');
module.exports = {
  ...
  ...
  plugins: [
    // 注册 Webpack 内置插件
    new webpack.DefinePlugin({
      VERSION: JSON.stringify("5fa3b9"),
      BROWSER_SUPPORTS_HTML5: true,
```

```
      TWO: "1+1",
      "typeof window": JSON.stringify("object")
    })
  ],
  ...
}
```

25.6.2　常用插件

下面介绍几款 Webpack 内置的常用插件的使用，插件的安装请参阅 25.6.1 节。

1. DefinePlugin

DefinePlugin 插件用来替换指定变量，代码示例如下：

```
// webpack.config.js
var webpack = require('webpack');
module.exports = {
  ...
  ...
  plugins: [
    new webpack.DefinePlugin({
      VERSION: JSON.stringify("5fa3b9"),
      BROWSER_SUPPORTS_HTML5: true,
      TWO: "1+1"
    })
  ],
  ...
}

// 待编译的文件
console.log("Running App version " + VERSION)// 编译后：console.log("Running App version
" + "5fa3b9")
if(!BROWSER_SUPPORTS_HTML5) require("html5shiv")// 编译后:if(!true) require("html5shiv")
var two = TWO// 编译后：var two = 1+1
```

2. ProvidePlugin

ProvidePlugin 可以自动加载当前模块依赖的其他模块并以指定别名注入到当前模块中。假如当前模块依赖 jquery 模块，同时我们想在模块中直接用“$”引用 jQuery 对象，但是不想手动 require jquery 模块。代码示例如下：

```
// 当前模块
$("#item")
```

此时我们只需要在 Webpack 配置文件中配置 ProvidePlugin 插件将 jquery 模块导出为 $ 变量即可。代码示例如下：

```
// webpack.config.js
var webpack = require('webpack');
module.exports = {
  ...
  ...
  plugins: [
    // 自动引入 jquery 模块并导出为 $ 变量，使各个模块可以直接通过“$”来引用 jQuery 对象
    new webpack.ProvidePlugin({
       $: "jquery"
    })
  ],
  ...
}
```

第 26 章 Rollup

“Webpack 2 输出的文件比起 Rollup 还是丑啊。但是 Rollup 针对非 JS 资源的插件生态不行，也没有热替换，不适合用在应用层。所以现阶段我还是用 Webpack 做应用层开发，用 Rollup 做库的打包。”

——尤小右

26.1 简介

在开发一个项目时，我们会将项目拆分成多个模块，每个模块完成相对独立的功能，我们可以很方便地单独开发代码。很可能存在这种情况：项目依赖很多第三方组件，依赖组件都很小，这对于浏览器来说是非常糟糕的，增加了许多请求；对于前端开发来说，严重影响了页面加载速度。这种情况必须规避。

针对上面情况解决办法有很多，大多都是采用模块化开发方式，使用模块化打包工具将所有文件最终打包到一个单独的输出文件中，大大减少了请求的数量。Browserify 和 Webpack 就是这样的打包工具。

使用这种打包工具很快、很好、很方便。但是我们是否注意到下面这样的问题：

```
var utils = require( 'utils' );
var query = 'Rollup';
utils.ajax( 'https://api.example.com?search=' + query ).then( handleResponse );
```

我们引入工具函数，但其实只想使用它的 ajax 方法，传统的打包方式是将所有代码全部打包，造成代码冗余。

ES 6 解决了这个问题，取代了引入全部工具函数，可以只引入所要使用的 ajax 方法。代码

示例如下：

```
import { ajax } from 'utils';
var query = 'Rollup';
ajax( 'https://api.example.com?search=' + query ).then( handleResponse );
```

可以很明显地体会到 Tree-shaking 的作用——bundle 文件中只保留了 utils 模块里的 ajax 方法。

另外，Tree-shaking 会抽取引用到的模块内容，将它们置于同一个作用域下，进而直接使用变量名就可以访问各个模块的接口；而不像 Webpack 那样在每个模块外还要包一层函数定义，再通过合并进去的 define/require 相互调用。

Rollup 是下一代 ES 6 模块打包工具。它采用 Tree-shaking 技术，利用 ES 6 模块静态分析语法树的特性，只将需要的代码提取出来打包，大大减小了代码体积。可以预见，未来的各种框架类库都会采用 ES 6 语法编写。

26.2 安装

全局安装，执行如下命令：

```
$ npm install -g rollup
```

另外，如果我们在 npm run script 环境中使用 Rollup 命令，则可以使用 npm I -D rollup，作为每一个项目的依赖。

26.3 配置

大多数选项都可以通过命令行直接指定，但是我们希望使用插件的配置文件，或者想以编程方式设置选项。

配置文件本身只是一个 JavaScript 模块（我们也可以使用 require 和 module.exports），代码示例如下：

```
import buble from 'rollup-plugin-buble';
export default {
  entry: 'src/main.js',
  dest: 'dist/bundle.js',
  format: 'umd',
  plugins: [ buble() ]
};
```

将会输出 umd 格式的内容到 dist/bundle.js 文件中。

注：通过命令行指定的参数选项将会覆盖配置文件中的相应选项，如果要实现上面例子中的配置输出，则可以在命令行中执行如下命令：

```
$ -c-f  umd -o dist/ bundle.cjs.js
```

Rollup 允许一个配置文件可以有多个目标文件，我们可以生成 umd 格式、ES 格式的文件，都是来自同一文件，我们无须重复工作。代码示例如下：

```
import babel from 'rollup-plugin-babel';
import babelrc from 'babelrc-rollup';

let pkg = require('./package.json');
let external = Object.keys(pkg.dependencies);

export default {
  entry: 'lib/index.js',
  plugins: [babel(babelrc())],
  external: external,
  targets: [
    {
      dest: pkg['main'],
      format: 'umd',
      moduleName: 'rollupStarterProject',
      sourceMap: true
    },
    {
      dest: pkg['jsnext:main'],
      format: 'es6',
      sourceMap: true
    }
  ]
};
```

结果目录如下：

```
|- dist
  |-rollup-starter-project.js
  |-rollup-starter-project.js.map
  |-rollup-starter-project.mjs.map
  |-rollup-starter-project.mjs.map
```

26.4　命令

在命令行中，运行 rollup -h 或 rollup -help 查看命令。

语法如下：

```
rollup [options] <entry file>
```

命令选项如下：

（1）-v

版本号。

```
$ rollup -v
rollup version 0.32.0
```

（2）-c

配置文件（默认为 rollup.config.js）。如果配置文件另有其名（例如 rollup.config.dev.js），在后面加上配置文件名即可。

```
$ rollup -c rollup.config.dev.js
```

（3）-w

监控文件变化后重新渲染文件。

（4）-i

全称：--input。

（5）-o

全称：--output。

（6）-f

格式化生成的包文件，全称：--format。支持如下参数：

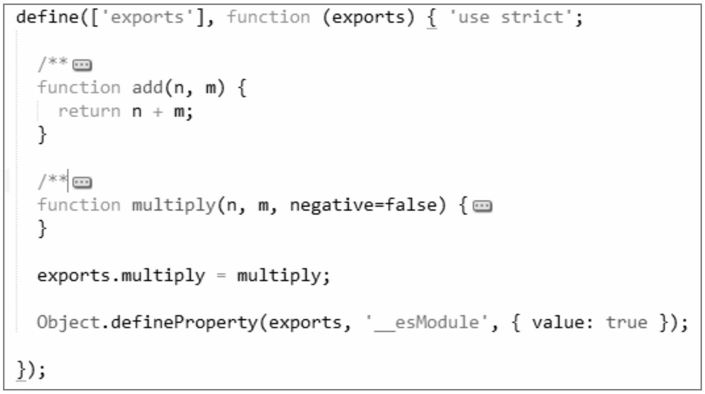
- amd —— 异步模块定义。目前，主要有两个 JavaScript 库实现了 AMD 规范，即 require.js 和 curl.js。
- cjs —— CommonJS 规范。

- es6（默认）—— ES 6 规范。
- iile —— “自执行”，放入<script>标签中。
- umd —— 通用模块定义。

将 index.js 转换为 amd 格式，执行如下命令：

```
$ rollup --format  amd -- lib/index.js > build/index.js
```

结果如图 26-1 所示。



```
define(['exports'], function (exports) { 'use strict';

  /**
  function add(n, m) {
    return n + m;
  }

  /**
  function multiply(n, m, negative=false) {
  }

  exports.multiply = multiply;

  Object.defineProperty(exports, '__esModule', { value: true });

});
```

图26-1　amd格式输出文件

（7）-e

全称：--external，扩展插件。

类型：

数组[Array]——外部依赖模块包的 ID 列表。ID 可以是：

- 外部依赖项的名称。
- resolved ID（例如一个文件的绝对路径）。

代码示例如下：

```
// app.js
import moment from 'moment';

setInterval(function() {
```

```
  var timeStr = moment().format('h:mm:ss a');
  console.log('the time is ' + timeStr);
}, 1000);

// build.js
import * as path from 'path';

rollup.rollup({
  entry: 'app.js',
  external: [
    'moment',
    path.resolve('./src/special-file.js')
  ]
}).then(...)
```

（8）-g

全称：--globals，全局属性，对 UMD/IIFE 模块有用。

类型：

Object :{ id: name}

代码示例如下：

```
var code = bundle.generate({
  format: 'iife',
  moduleName: 'MyBundle',
  globals: {
    backbone: 'Backbone',
    underscore: '_'
  }
}).code;
```

（9）-n

全称：--name，UMD/IIFE 模块的名称。

代码示例如下：

```
var code = bundle.generate({
  format: 'iife',
```

```
  moduleName: 'MyBundle'
}).code;
// -> var MyBundle = (function () {...
```

（10）-u

全称：--id，AMD/UMD 模块的 id。

代码示例如下：

```
var code = bundle.generate({
  format: 'amd',
  moduleId: 'my-bundle'
}).code;
// -> define(['my-bundle'],...
```

（11）-m

全称：--sourcemap，产生 sourcemap。

（12）--no-strict

禁止生成“use strict”格式的代码。

（13）--no-indent

禁用缩进。

（14）--environment <values>

设置传递<values>给配置文件。

注：如果使用--environment 选项，通过 process.env 参数调用。

代码示例如下：

```
// using the example above
process.env.INCLUDE_DEPS === 'true' // always a string
process.env.BUILD === 'production'
```

（15）--no-conflict

生成 UMD 全局 noConflict 方法。

（16）--intro

在 bundle 文件最前面插入内容。

（17）--outro

在 bundle 文件最后面插入内容。

（18）--banner

在 bundle 文件最前面插入内容。

（19）--footer

在 bundle 文件最后面插入内容。

26.5　插件

Rollup 也支持使用插件，写到配置对象的 plugin 中即可。这里以 rollup-plugin-babel 为例，代码示例如下：

```
import babel from 'rollup-plugin-babel';
export default {
  entry: 'src/main.js',
  format: 'cjs',
  plugins: [ babel() ],
  dest: 'rel/bundle.js'
};
```

与 Webpack 不同的是，babel 的预设不像 Webpack 那样可以直接写在配置文件中，而是独立写个“src/.babelrc”（注意，我们可以写在 src 下，而不是非得放在项目根目录下）：

```
{
  "presets": ["es2015-rollup"]
}
```

注：在使用 babel 插件前，首先确保安装了 rollup-plugin-babel 和 babel 预设 babel-preset-es2015-rollup：

```
$ npm i rollup-plugin-babel babel-preset-es2015-rollup
```

这时候就能配合 babel 把 ES 6 模块编译成 ES 5 的 bundle 了。

其他常见插件列表如下：

- alias —— 定义打包时所用的别名。
- ascii —— 在字符串中重写非 ASCII 字符。
- auto-transform —— 根据 package.json 中的键来应用 Browserify 自动转换，就像 Browserify 做的一样。
- babel —— 用 Babel 翻译代码。
- bower-resolve —— 在 Rollup 中使用 Bower 解决算法。
- browserify-transform —— 将 Bowserify 作为插件使用进行转换。
- buble —— 用 Bublé（与 Babel 相似，但是比其快得多）翻译代码。
- coffee-script —— 将 coffeeScript 代码转换为 JavaScript 代码。
- commonjs —— 将 CommonJS 模块转换为 ES 6 形式。
- eslint —— 核实入口以及引入的脚本代码。
- filesize —— 在命令行中显示打包的文件大小。
- hypothetical —— 从假想的文件系统中引入模块。
- image —— 引入 JPG、PNG、GIF 以及 SVG 图片。
- includepaths —— 提供引入模块的基础路径。
- inject —— 检测依赖并且注入之。
- istanbul —— 使用 Istanbul 来处理代码覆盖。
- json —— 将 JSON 转换为 ES 6。
- jst —— 编译模板文件。
- jsx —— 编译 React 的 JSX，以及其他类似 JSX 的组件。
- memory —— 从内存中读取入口文件。

- multi-entry —— 允许有多个入口文件，而不是仅有一个。
- node-builtins —— 允许在 Rollup 中使用 Node.js 内置的包。
- node-resolve —— 使用 Node.js 模块解决方案（比如，使用以 npm 方式安装的来自 node_modules 的模块）。
- pegjs —— 引入 PEG.js 语法作为语法解析。
- postcss —— 编译 PostCSS 并且插入 head 中。
- ractive —— 预编译 Ractive 组件。
- replace —— 替换一组字符串的出现。
- riot —— 编译 Riot.js 标签文件。
- string —— 将文本文件以 string 形式引入。
- strip —— 移除调试时使用的陈述及函数，比如 console.log。
- stylus-css-modules —— 编译 Stylus 并且注入 CSS 模块。
- typescript —— 将 TypeScript 编译为 JavaScript。
- uglify —— 减小生成的 bundle 的体积。
- vinyl —— 从 Vinyl 文件中进行引入。
- vue —— 编译 Vue 组件。

26.6 常见问题解析

- 我们可以使用没有 ES 6 的依赖吗？

可以，Rollup 在 CommonJS 模块下 Tree-shaking 是不能工作的，但是我们可以通过插件将它们转换成 ES 6 模块。

第 27 章 Browserify

27.1 安装

全局安装，执行如下命名：

```
$ npm i -g browserify
```

除了全局安装，我们也可以将 Browserify 作为项目依赖在项目中进行安装，执行如下命令：

```
$ npm i browserify --save-dev
```

27.2 基本使用

与 Node.js 支持的 CommonJS 规范一样，Browserify 通过 require 来加载依赖文件。假设有如下目录结构：

```
example
  |- concat.js
  |- log.js
  |- index.js
  |- index.html
```

各个文件内容如下：

```
// 入口文件，加载相应的依赖文件
var log = require('./log.js');
var concat = require('./concat.js');
var arr = ['Just', 'A', 'Browserify', 'Demo', '!'];
log(concat(arr));
```

```
// log.js
module.exports = function (strToLog) {
  console.log(strToLog);
}

// concat.js
module.exports = function (arr) {
  return arr.join(' ');
}

<!-- 入口 HTML 文件 -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>browserify build demo</title>
</head>
<body>
  <!-- 引入打包后的 JS 文件 -->
  <script src="bundle.js"></script>
</body>
</html>
```

在 exmpale 目录下执行以下命令进行打包：

```
$ browserify index.js > bundle.js
```

接下来我们在浏览器中打开 index.html 文件，可以看到控制台输出如下：

```
Just A Browserify Demo !
```

27.3　转换模块

有时候直接 require 源文件并不能达到我们的目的，我们需要对文件进行预处理后再输出。这时可以利用 Browserify 提供的转换模块机制，转换模块可以对源文件进行相应的处理后再输出到最终文件中。

27.3.1　安装转换模块

转换模块即遵循一定规则的 npm 模块，所以我们直接通过 npm 来安装。这里以 envify 转

换模块为例：

（1）全局安装

```
$ npm install -g envify
```

（2）本地安装

```
$ npm install envify –save-dev
```

27.3.2　使用转换模块

1. 配置方式

在 package.json 文件的 Browserify 字段指定转换模块，Browserify 编译时会自动依次调用 transform 数组内指定的转换模块：

```
"browserify": { "transform": [ "envify" ] }
```

2. 参数方式

除了在 package.json 中指定转换模块外，还可以在命令行中进行指定：

```
$ browserify entry.js -t envify> bundle.js
```

我们也可以向转换模块传入参数，以下命令向 envify 转换模块传入 NODE_ENV 参数，值为 development：

```
$ browserify index.js -t [ envify --NODE_ENV development ] > bundle.js
```

27.3.3　相关转换模块介绍

Browserify 生态中提供了很多转换模块供我们实现各类需求。本节介绍和 Vue.js 开发相关的转换模块。

1. vueify

vueify 可以让我们将组件的模板、样式、JS 逻辑写在同一个文件中。

（1）安装

```
$ npm install vueify --save-dev
```

如果使用 npm 3 及以上版本，需要手动安装 babel 依赖：

```
$ npm installbabel-core babel-preset-es2015 babel-runtime babel-plugin-transform-runtime
--save-dev
```

使用该转换模块允许我们通过以下形式来写 Vue 组件，代码示例如下：

```
// app.vue
<style>
  .red {
    color: #f00;
  }
</style>

<template>
  <h1 class="red">{{msg}}</h1>
</template>

<script>
export default {
  data () {
    return {
      msg: 'Hello world!'
    }
  }
}
</script>
```

也可以使用 lang 来指定预处理器，代码示例如下：

```
// app.vue
<style lang="stylus">
  .red
  color #f00
</style>

<template lang="jade">
  h1(class="red") {{msg}}
</template>

<script lang="coffee">
  module.exports =
  data: ->
    msg: 'Hello world!'
</script>
```

还可以使用 src 引入外部文件，代码示例如下：

```
<style lang="stylus" src="style.styl"></style>

// main.js，入口 JS 文件
var Vue = require('vue')
var App = require('./app.vue')

new Vue({
  el: 'body',
  components: {
    app: App
  }
})

<!-- 入口 HTML 文件 -->
<body>
  <app></app>
  <script src="build.js"></script>
</body>
```

（2）编译

```
$ browserify -t vueify -e src/main.js -o build/build.js
```

2. envify

envify 转换模块用来替换代码中的有关 node 环境变量代码片段。这样我们就可以只在开发环境中输出调试信息，在生产环境中由于条件判断为 false，调试信息不会输出。还可以结合 uglifyify 库在生产环境中将调试信息直接移除。

（1）安装

```
$ npm install envify browserify
```

（2）代码示例

```
// index.js
if (process.env.NODE_ENV === "development") {
  console.log('development only')
}
```

假设当前 NODE_ENV 环境变量值为 production，运行转换模块后将会得到：

```
// bundle.js
if ("production" === "development") {
  console.log('development only')
}
```

（3）编译

通过 Browserify 命令行使用 envify 转换模块：

```
$ browserify index.js -t envify > bundle.js
```

还可以在编译时传入自定义环境变量：

```
$ browserify index.js -t [ envify --NODE_ENV development ] > bundle.js
$ browserify index.js -t [ envify --NODE_ENV production  ] > bundle.js
```

第 28 章 vue-loader

vue-loader 是基于 Webpack 的 loader，在 Vue 组件化中起着决定性作用。

28.1 如何配置

vue-loader 的配置和 Webpack 其他 loader 的配置类似，对.vue 后缀增加处理。

配置如下：

```
module.exports = {
  entry: {
    app: './src/main.js'
  },
  module: {
    loaders: [
      {
        test: /\.vue$/,
        loader: 'vue'
      }
    ]
  }
}
```

其实配置还是非常简单、直观的。

28.2 包含内容

我们来看.vue 文件的组成。其一般包含：

- template 标签——包裹 HTML 模板片段。

- script 标签——配置 Vue 和载入其他组件或者依赖库。
- style 标签——设置样式

代码示例如下：

```
<template>
  <div id="app">
    <p> Welcome to DDFE,We love Vue.js! </p>
  </div>
</template>

<script>
  export default {
  }
</script>

<style>
html {
  height: 100%;
}
</style>
```

28.3　特性介绍

1. template

- 支持 lang 配置多种模板语法。
- 只支持单个 template 标签。

2. style

- 支持 lang 配置多种预编译语法。
- 支持 scope 属性，这样 CSS 只应用到当前组件的元素中，类似于 Shadow DOM，但是不需要任何插件来支持。
- 一个.vue 文件中可以包含多个 style 标签。

- 内置 PostCSS 和 autoprefixer 来自动添加浏览器前缀。

3. script

- 默认支持 Babel（使用 babel-loader）来编译 ES 6 语法糖。
- 对于 7.0 及以上版本，使用 Babel 6；如果使用 Babel 5，则需要使用 6.x 版本。
- 支持通过 import 方式载入其他.vue 后缀的组件文件。
- 只支持单个 script 标签。

4. Hot Reload

.vue 文件修改后，默认支持对应的页面自动刷新。

默认内置 loader 的配置如下：

```
var defaultLoaders = {
  html: 'vue-html-loader',
  css: 'vue-style-loader!css-loader',
  js: 'babel-loader?presets[]=es2015&plugins[]=transform-runtime&comments=false'
}
```

我们可以看到：

- JS 默认使用 babel-loader 编译，加了一些参数。
- HTML 默认使用 vue-html-loader。
- CSS 默认使用 vue-style-loader 和 css-loader。

28.4 常见问题解析

1. 如何自定义 autoprefixer

```
<!-- webpack.config.js -->
module.exports = {
  //. . .
  vue: {
    autoprefixer: false
  }
}
```

2. 如何自定义 PostCSS

```
<!-- webpack.config.js -->
module.exports = {
  //. . .
  vue: {
    postcss:{
     //. . .
    },
  }
}
```

3. 如何自定义 style 的预编译语言

```
<!-- 第一步使用 lang 配置 -->
<style lang="sass">
</style>

<!-- 第二步安装依赖 -->
npm install sass-loader node-sass
```

4. 如何自定义 loaders

```
module.exports = {
  module: {
    loaders: [
      {
        test: /\.vue$/,
        loader: 'vue'
      }
    ]
  },
  vue: {
    loaders: {}
  }
}
```

28.5　源码解析

我们先来看一下 vue-loader 文件结构，如图 28-1 所示。

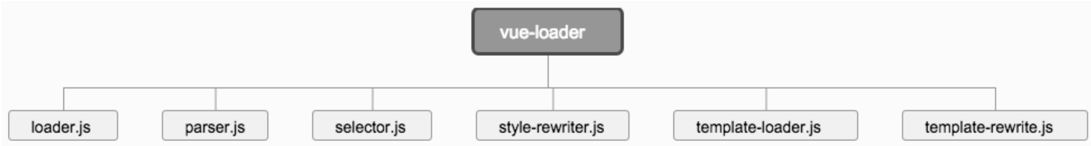


图28-1　vue-loader文件结构

- 主入口文件：loader.js。接受.vue 文件的全部内容，然后把内容 content、文件名 fileName 传递下去。
- 内容解析文件：parser.js。首先通过调用 hash 函数把文件名和文件内容生成唯一的 key，然后通过 cache 检测，如果命中 cache，就直接返回；如果没有命中，则调用 parse5 的 parseFragment 方法对文件内容进行解析，遍历子节点内容（只处理 template、style 和 script 节点），取出对应的特性属性（lang、src 和 scoped），针对空节点的 script 和 style 也进行了优化。parser.js 解析过程如图 18-2 所示。

我们来看一个最简单的.vue 文件。

```
<template>
  <div id="app">
    <p>
      Welcome to DDFE,We love Vue.js!
    </p>
  </div>
</template>

<script>
  export default {
  }
</script>

<style>
html {
  height: 100%;
}
</style>
```

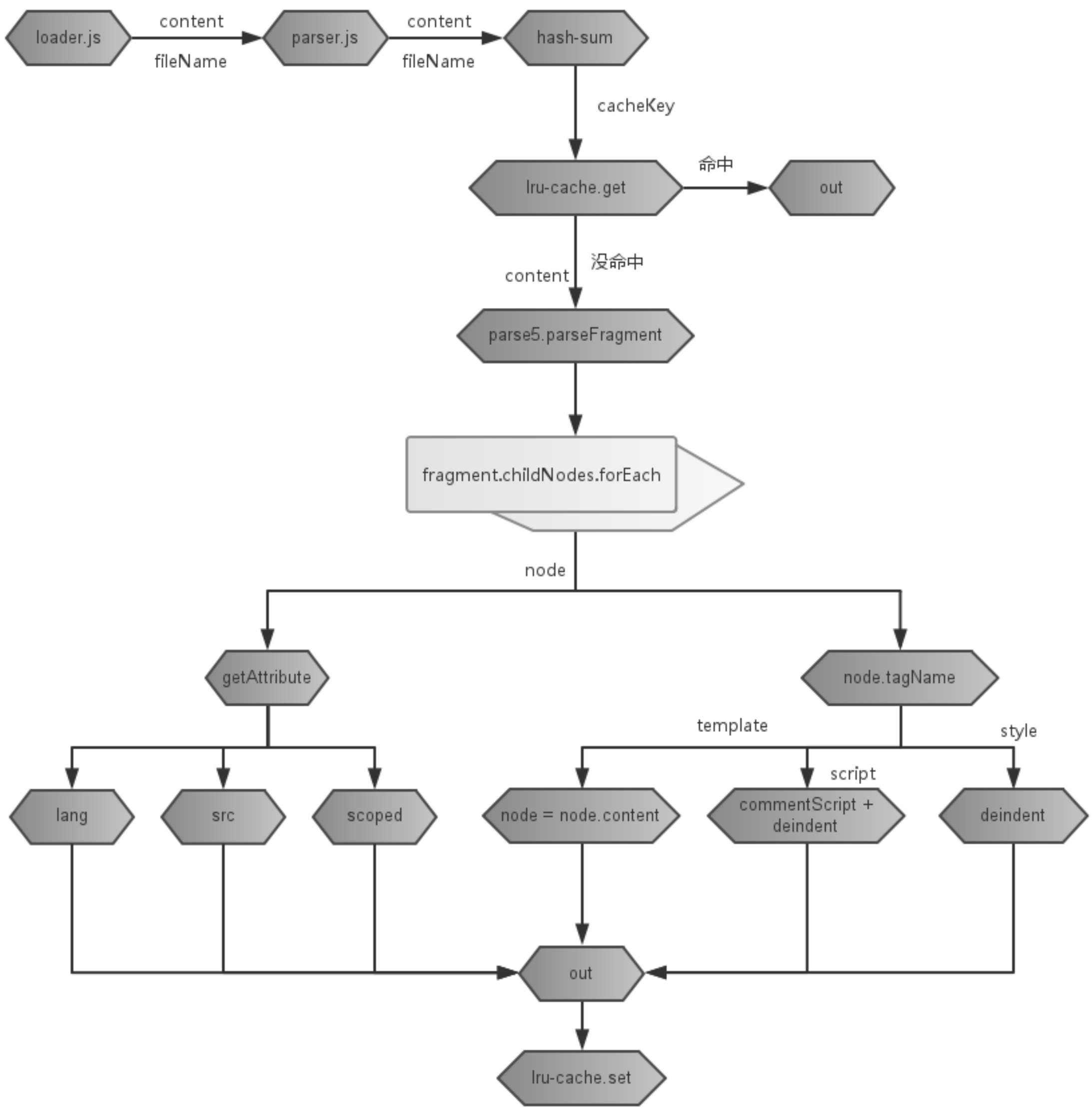


图28-2　parser.js解析过程

通过 parse5.parseFragment 解析后：

```
{ nodeName: '#document-fragment',
  quirksMode: false,
  childNodes:
   [ { nodeName: 'template',
```

```
      tagName: 'template',
      attrs: [],
      namespaceURI: 'http://www.w3.org/1999/xhtml',
      childNodes: [],
      parentNode: [Circular],
      content: [Object],
      __location: [Object] },
    { nodeName: '#text',
      value: '\n\n',
      parentNode: [Circular],
      __location: [Object] },
    { nodeName: 'script',
      tagName: 'script',
      attrs: [],
      namespaceURI: 'http://www.w3.org/1999/xhtml',
      childNodes: [Object],
      parentNode: [Circular],
      __location: [Object] },
    { nodeName: '#text',
      value: '\n\n',
      parentNode: [Circular],
      __location: [Object] },
    { nodeName: 'style',
      tagName: 'style',
      attrs: [],
      namespaceURI: 'http://www.w3.org/1999/xhtml',
      childNodes: [Object],
      parentNode: [Circular],
      __location: [Object] },
    { nodeName: '#text',
      value: '\n',
      parentNode: [Circular],
      __location: [Object] }
  ]
}
```

实现 template 支持其他模板引擎，比如 jade：

```
<template lang="jade">
  #app
     p
```

```
      | Welcome to DDFE,We love Vue.js!
</template>
```

template-loader.js 解析过程如下：

接受模板 template 的内容，然后前面传递了{ raw: true, engine: 'jade' }的 query 参数对象，如果 engine 没有设置，则直接用文件的后缀（比如这里是 App.vue，取后缀就是 vue）。模板的解析还是依赖 consolidate 这个核心包来判断是否支持这类模板和后续的 render。

这里面会检查是否安装了对应的依赖包，比如依赖 jade，如果没有安装 consolidate.js 会在后台抛错，如图 28-3 所示。

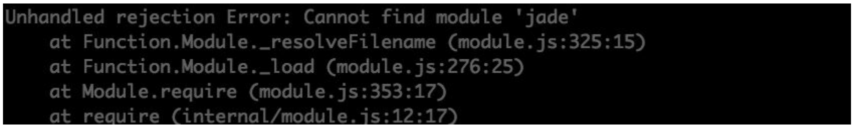


图28-3　consolidate.js寻找依赖jade抛错

style-rewriter.js 解析过程如下：

接受模板 style 的内容，然后前面传递了{ id: '_v-11f8ff75', scoped: true }的 query 参数对象，里面会判断 autoprefixer 的配置，调用 object-assign 把它的配置和用户在 Webpack 中配置 options 的 autoprefixer 合并，最终还是依赖 PostCSS 来处理。

```
<!-- autoprefixer -->
var options = this.options.vue || {}
var autoprefixOptions = options.autoprefixer
// 默认自动加载，触发手动在配置文件中配置 Vue 对象中的 autoprefixer
if (autoprefixOptions !== false) {
  autoprefixOptions = assign(
    {},
    // also respect autoprefixer-loader options
    this.options.autoprefixer,
    autoprefixOptions
  )
  var autoprefixer = require('autoprefixer')(autoprefixOptions)
  plugins.push(autoprefixer)
}
```

我们重点看一下配置 scoped 的处理实现。

```
<!-- App.vue -->
<template>
<div id="app">
<p>Welcome to DDFE,We love Vue.js!</p>
  </div>
    </template>
  <style scoped>
#app {
  text-align: center;
}
</style>
```

编译后：

```
<div id="app" _v-11f8ff75>
  <p _v-11f8ff75>Welcome to DDFE,We love Vue.js!</p>
</div>
#app[_v-11f8ff75] {
  text-align: center;
}
```

源码如下：

```
<!-- loader.js 中提到了 id 的生成规则 -->
var hash = require('hash-sum')
var filePath = this.resourcePath
var moduleId = '_v-' + hash(filePath)

<!-- style-rewriter.js -->
var addId = postcss.plugin('add-id', function (opts) {
  return function (root) {
    root.each(function rewriteSelector (node) {
      if (!node.selector) {
        // handle media queries
        if (node.type === 'atrule' && node.name === 'media') {
          node.each(rewriteSelector)
        }
        return
      }
```

```
      node.selector = selectorParser(function (selectors) {
        selectors.each(function (selector) {
          var node = null
          selector.each(function (n) {
            if (n.type !== 'pseudo') node = n
          })
          selector.insertAfter(node, selectorParser.attribute({
            attribute: opts.id
          }))
        })
      }).process(node.selector).result
    })
  }
})

if (query.scoped) {
  plugins.push(addId({ id: query.id }))
}
```

28.6　工具包介绍

上面我们已经陆续提到了一些 vue-loader 中使用的不错的第三方工具包。

- PostCSS，主要用来处理.vue 文件中 style 部分的一些特性解析，比如 scoped 等。
- autoprefixer，其实它就是 PostCSS 最流行的插件，使用 caniuse 站点的数据自动给一些 CSS 规则添加浏览器前缀。
- postcss-selector-parser，提供一些 API 来解析选择器。
- source-map，生成 sourcemap 的工具包。
- vue-template-validator，处理 Vue.js 模板 template 在编译器中的一些错误。
- consolidate，模板引擎合集，支持市面上基本所有的模板引擎，比如 jade、doT.js、ejs 等。
- parse5，HTML 语法解析工具。

- object-assign，来自大名鼎鼎的 sindresorhus 作品，一个 ES 5 Object.assign()的 polyfill，不过不覆盖原生方法。
- lru-cache，支持 least-recently-used 算法的 cache 工具包。
- hash-sum，非常快速的唯一 hash 值生成器。
- de-indent，从代码块中删除多余的 indent。
- loader-utils，Webpack loader 的依赖库，提供很多常用的方法，比如 parseQuery 等。

第 29 章 PostCSS

PostCSS 是一个用 JavaScript 插件来转换 CSS 的工具，目前已经有 200 名插件，这些插件可以 lint CSS，支持变量、mixins、内联的图片等。

简单来说，PostCSS 可以将 CSS 转换为 JavaScript 能够处理的数据格式，基于 JavaScript 所写的插件可以完成上述各种操作。PostCSS 为这些插件提供了接口，方便其完成各自的功能，但是不会对 CSS 代码做任何修改。从理论上讲，PostCSS 的插件可以对 CSS 进行任何操作，只要我们有需求，就可以写一个 JavaScript 插件来实现。

29.1 安装

PostCSS 针对不同的构建工具提供了不同的安装工具。在 Webpack 中该工具名为 postcss-loader，全局安装方式如下：

```
$ npm install -g postcss-loader
```

除此之外，当需要用到 PostCSS 的插件时，也可以使用 npm 安装。比如，大名鼎鼎的 autoprefixer 安装方式如下：

```
$ npm install autoprefixer
```

29.2 配置

PostCSS 一般与 Gulp、Webpack 等构建工具搭配使用。在 vue-loader 中使用 PostCSS 时，需要在 webpack.config.js 中进行配置。当需要使用 PostCSS 的插件时，在 vue 选项中向 postcss

设置选项传入一个数组，比如使用 CSSNext 插件的配置代码示例如下：

```
// webpack.config.js
module.exports = {
  // other configs...
  vue: {
    // use custom postcss plugins
    postcss: [require('postcss-cssnext')()],
    // disable vue-loader autoprefixing.
    // this is a good idea since cssnext comes with it too.
    autoprefixer: false
  }
}
```

除了提供一个数组用于存放引用的数组外，postcss 设置选项还可以接受：

- 一个可以返回插件数组的函数。代码示例如下：

```
postcss: function () {
  return [precss, autoprefixer];
}
```

- 一个对象，该对象包含将被传给 PostCSS 处理器的设置选项。代码示例如下：

```
postcss: {
  plugins: [...], // list of plugins
  options: {
    parser: sugarss // use sugarss parser
  }
}
```

29.3　命令

在命令行或者 npm scripts 中使用 PostCSS 需要额外安装 postcss-cli，其安装方式如下：

```
$ npm install postcss-cli
```

语法如下：

```
postcss [options] [-o output-file|-d output-directory|-r] [input-file]
```

命令选项如下：

（1）--output|-o

指定输出文件。如果没有指定输出文件，则 PostCSS 会写入到 stdout，但是对输出文件位置有依赖的插件将无法正确工作。

相似的，如果输入文件没有指定，PostCSS 将会从 stdin 读入，对输入文件位置有依赖的插件将无法正确工作。

（2）--dir|-d

指定多个输出文件的位置。需要指定--output、--dir 或者--replace 选项，但不是三者都需要被指定。在提供了多个输入文件的情况下需要使用--dir 或者--replace。

（3）--replace|-r

用生成的输出文件替换单个或者多个输入文件。需要指定--output、--dir 或者--replace 选项，但不是三者都需要被指定。在提供了多个输入文件的情况下需要使用--dir 或者--replace。

（4）--use|-u

指定要使用的插件。可以指定多个插件，使用--use 选项或者在 config 文件中至少需要指定一个插件。

（5）--map|-m

激活生成 sourcemap。默认设置是生成行内映射。如果要在其他的.map 文件中生成 sourcemap，使用--mapfile 或者--no-map.inline 命令。

还可以使用更多的 sourcemap 选项，例如：

- --no-map —— 不要生成 sourcemap，即使之前已有 map 存在。
- --map.annotation <path> —— 指定可选的用于附加在 CSS 后面的 sourcemap 注解的路径。
- --no-map.annotation —— 禁止添加 CSS 的注解。
- --no-map.sourcesContent —— 从 map 中移除原始的 CSS。

（6）--local-plugins

从当前工作目录的 node_modules 文件夹开始寻找插件。如果没有这个选项，postcss-cli 将

会从其安装位置的 node_modules 文件夹开始寻找插件——确切地说，在 postcss-cli 是全局安装的情况下，它将会寻找全局安装的插件。

（7）--watch|-w

监视文件系统的改变并且当源文件更改时重新编译。

当行内 CSS 被引入时，将向 JavaScript 配置文件中添加一个更新处理器，来保证引用的模块被纳入考量。代码示例如下：

```
{
  "postcss-import": {
    onImport: function(sources) {
      global.watchCSS(sources, this.from);
    }
  }
}
```

对 postcss-import 来说，该处理器会被自动添加。

（8）--config|-c

指定内容为插件配置的 JSON 文件，插件的名称作为键。代码示例如下：

```
{
  "autoprefixer": {
    "browsers": "> 5%"
  },
  "postcss-cachify": {
    "baseUrl": "/res"
  }
}
```

如果函数允许作为插件的参数，那么 JavaScript 配置也可以使用。代码示例如下：

```
module.exports = {
  "postcss-url": {
    url: function(url) { return "http://example.com/" + url; }
  },
  autoprefixer: {
    browsers: "> 5%"
  }
};
```

可选的配置选项可以作为--plugin.option 的参数。

注意，命令行选项也可以在配置文件中进行指定。代码示例如下：

```
{
  "use": ["autoprefixer", "postcss-cachify"],
  "input": "screen.css",
  "output": "bundle.css",
  "local-plugins": true,
  "autoprefixer": {
    "browsers": "> 5%"
  },
  "postcss-cachify": {
    "baseUrl": "/res"
  }
}
```

（9）--syntax|-s

将可选的模块作为定制的 PostCSS 语法使用。

（10）--parser|-p

将可选的模块作为定制的 PostCSS 输入解析器使用。

（11）--stringifier|-t

将可选的模块作为定制的 PostCSS 输出转换器使用。

（12）--help|-h

显示帮助信息。

29.4　插件

- autoprefixer，PostCSS 最知名的插件，其作用是为 CSS 中的属性添加浏览器特定的前缀。
- cssnext，该插件允许开发人员在当前项目中使用 CSS 将来版本中可能会加入的新特性。cssnext 负责把这些新特性转译成在当前浏览器中可以使用的语法。在这些新特性中也包含了 autoprefixer 的功能，所以当使用了 cssnext 后就无须再使用 autoprefixer 了。

- precss，包含了能够使用类似 Sass 中的变量、嵌套、混合等功能的插件。
- postcss-sorting，对 CSS 内容按照指定规则排序。
- short，添加并且扩展很多的简写属性。
- postcss-use，在当前样式表中直接使用 PostCSS 的插件。
- postcss-assets，插入图像尺寸以及内联文件。
- postcss-sprites，生成图像雪碧图。
- font-magician，生成所有 CSS 中需要的@font-face 规则。
- postcss-inline-svg，内联 SVG 图片并且定制其样式。
- postcss-write-svg，直接在 CSS 中书写简单的 SVG。
- stylelint，模块化的 CSS 样式检查器。
- stylefmt，根据 stylelint 自动格式化 CSS 的工具。
- doiuse，根据 Can IUse 的数据来检测 CSS 的浏览器支持度。
- colorguard，帮助开发者维护一个统一的调色板。

第 30 章

拓展篇

30.1 Composition Event

Composition Event，中文译为“复合事件”，是 DOM 3 级事件中新添加的一类事件类型，用于处理 IME 的输入序列。IME（Input Method Editor，输入法编辑器）可以让用户输入在物理键盘上找不到的字符。复合事件就是针对检测和处理这种输入而设计的。因为以上所述原因，复合事件很少被拉丁系语言输入的开发者所知（因为拉丁字母都能通过物理键盘输入）。当然，即使是使用非拉丁系语言比如中文作为输入的开发者，也不见得了解复合事件，因为在开发中用到该种事件类型的情况比较少见。

IME 复合系统的工作原理是：缓存用户的键盘输入，直到一个字符被选中后才确定输入。缓存的键盘输入会暂时展示在输入框中，但不会真正被插入到 DOM 中，如图 30-1 所示。但是如果在复合事件的过程中改变了输入框的值（比如切换了输入法或者直接按下 Enter 键），复合事件将提前结束，同时缓存的键盘输入值将会插入到输入框中。



图30-1　缓存键盘输入

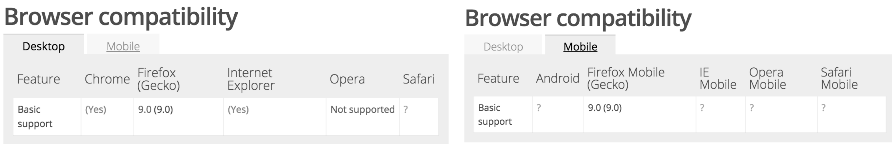
复合事件类型包含以下几种事件：

- compositionstart —— 在 IME 的文本复合系统打开时触发。
- compositionend —— 在 IME 的文本复合系统关闭即用户选中了字符并确定输入时触发，表示返回正常键盘的输入状态。

- compositionupdate —— 在 compositionstart 事件触发后、compositionend 事件触发前这段时间内，每次向输入字段中进行输入时均会触发。

注：input 事件将在复合事件后触发。

但是，实际情况与理想还是有一定距离的，复合事件的兼容性比较一般。如图 30-2 所示是 MDN 中列出的兼容性表现，详情可参见 https://developer.mozilla.org/en-US/docs/Web/API/CompositionEvent。



Browser compatibility

Desktop | Mobile

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	(Yes)	9.0 (9.0)	(Yes)	Not supported	?

Browser compatibility

Desktop | Mobile

Feature	Android	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile
Basic support	?	9.0 (9.0)	?	?	?

图30-2　复合事件的兼容性表现

综上所述，在使用复合事件处理 input 相关问题时，仍然需要慎重。

30.2　ES 6

ECMAScript 6（以下简称 ES 6）是 JavaScript 语言的最新一代标准，发布于 2015 年 6 月，因为 ECMA 委员会决定从 ES 6 起每年更新一次标准，因此 ES 6 被改名为 ES 2015，后面的标准将按照发布年份命名，如 ES 2016、ES 2017 等。

关于 ECMAScript 与 JavaScript 的关系，简单来说，前者是后者的规格，后者是前者的一种实现。通俗点说，就是后者是前者的一种方言。在通常情况下，二者之间可以划等号。

ES 6 是 JavaScript 的一次极为重大的更新，引入了很多新特性，这些特性解决了 JavaScript 长久以来被开发者所诟病的许多缺点。同时，ES 6 也给 JavaScript 的语法带来了重大变革，它们使 JavaScript 变得更加强大、更富有表现力。尽管属于重大升级，但 ES 6 仍然秉持了最大化兼容已有代码的设计理念，因此采用 ES 5 及之前标准编写的 JavaScript 代码在 ES 6 的环境下将继续正常运行。

由于发布日期尚短，当前主流的浏览器对 ES 6 还没有全面支持，不过好消息是支持程度正在逐渐提高，目前在各大浏览器的最新版本中 ES 6 的大部分特性都已经实现。而 Node.js 对 ES 6 的支持性还要优于浏览器，通过 node 可以体验更多 ES 6 的特性。我们也可以使用 Babel 等 JavaScript 预编译器将 ES 6 代码转换为 ES 5 代码，从而在现有环境中执行。

下面将简单介绍 Vue.js 源码中大量采用的 ES 6 的新特性——模块、let 和 const。

30.2.1　模块

历史上，JavaScript 一直没有模块的概念，这会导致当项目大到一定程度时 JavaScript 代码将变得难以复用且极难维护。因此，在相当长一段时间内，对于大型复杂项目来说，JavaScript 基本从一开始就被排除在方案之外。环顾其他开发语言，如 Python 的 import、Ruby 的 require 等都确保了模块功能的实现，唯独 JavaScript 在官方标准上一直缺少对模块的定义，直到 ES 6 的出现。

在 ES 6 发布之前，开发者社区就制定了一些模块加载的方案，其中使用最广泛的是 CommonJS（node 模块化加载）规范和 AMD（RequireJS）规范。前者用于服务器环境，后者则专注于浏览器环境。

如今，ES 6 从官方标准中带来了模块化开发规范。下面主要介绍 ES 6 模块化开发当中最重要的 export 和 import 概念。

1. export

在 ES 6 中，一个文件就是一个模块，一个模块内部的所有变量，对于外部来说是无法获取的，除非使用关键词 export 对外暴露接口，暴露的各接口通过名字来进行区分。如以下示例代码，lib.js 模块通过 sqrt、square、diag 向外界暴露三个接口。

```
//------ lib.js ------
/*
暴露三个接口给外界
*/
export const sqrt = Math.sqrt;
export function square(x) {
  return x * x;
}
export function diag(x, y) {
  return sqrt(square(x) + square(y));
}
```

export 也可以采用下面的方式暴露接口：

```
//------ lib.js ------
const sqrt = Math.sqrt;
```

```
function square(x) {
  return x * x;
}

function diag(x, y) {
  return sqrt(square(x) + square(y));
}

// 通过 export 暴露接口的第二种语法，使用大括号指定要暴露的接口
export {sqrt, square, diag};
```

建议采用第二种方式，因为其结构清晰，模块暴露了哪些接口一目了然。

在通常情况下，export 暴露的接口就是其本来的名字，不过可以采用 as 语法进行别名 export，这种导出方式可以将一个接口通过 *n* 个名字对外暴露。代码示例如下：

```
//------ lib.js ------
const sqrt = Math.sqrt;
// 通过两个别名对外界暴露
export {sqrt as sq1, sqrt as sq2};
```

注：在 ES 6 模块规范中，如果 b 模块从 a 模块导入一个原始值后，在 a 模块中修改了这个原始值，那么 b 模块中的值也会与 a 模块中最新的值保持同步，即 export 暴露的接口与其在模块内部对应的值是一种动态绑定的关系，通过接口可以获取模块内部实时的值。

2. import

使用 export 命令对外暴露接口以后，其他 JavaScript 文件就可以通过 import 命令加载这个模块（文件）。在 main.js 模块中就可以通过 import 引入上一节中 lib.js 暴露的接口，代码示例如下：

```
//------ main.js ------
/*
通过 import 语法从 lib 模块中导入所需要的接口
注意大括号中的接口名必须在 lib.js 模块中
通过 export 关键词导出
*/
import { square, diag } from './lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

import 也可以采用 as 语法对引入的变量重命名，代码示例如下：

```
//------ lib.js -----
export var myVar1 = 'var1';
```

```
//----- main.js -----
import {myVar1 as myCustomVar1} from './lib';
console.log(myCustomVar1);
```

import 会执行加载的模块，因此有空 import 的语法。代码示例如下：

```
// 只加载执行模块，不引用任何接口
import 'lib'
```

import 还可以整体加载模块，达到命名空间的效果。代码示例如下：

```
//------ lib.js -----
export var myVar1 = ···;
export let myVar2 = ···;
export const MY_CONST = ···;

export function myFunc() {
  ...
}
export function* myGeneratorFunc() {
  ...
}
export class MyClass {
  ...
}

//----- main.js -----
import * as lib from './lib';
console.log(lib.myVar1);
console.log(lib.myVar2);
new lib.MyClass();
```

3. export default

上面使用的 export 有一个小问题，即使用模块接口的人必须要知道该模块 export 了哪些接口。有时候一个模块实际上只对外暴露一个接口，这时候实际上没必要再限定暴露的接口名字，那么在 ES 6 中可以使用 export default 语法让模块调用者自定义要导入的接口名字。代码示例如下：

```
//------ myFunc.js ------
export default function () {}
```

```
//------ main1.js ------

/*
注意：myFunc 不能包含在{}里；myFunc 可以替换为任意喜欢的名字
*/
import myFunc from 'myFunc';
myFunc();
```

本质上，export default 就是输出一个名为 default 的变量或方法，然后系统允许我们进行重命名。代码示例如下：

```
// lib.js
function add(x, y) {
  return x * y;
}
export {add as default};

// 等同于
// export default add;

// main.js
import { default as myAdd } from 'lib';

// 等同于
// import myAdd from 'lib';
```

使用 export default 命令，我们可以很直观地引入模块，比如引用 jquery 模块时可以这样写：

```
import $ from 'jquery';
```

4. export/import 在 Vue.js 中的使用

Vue.js 采用 export/import 进行模块化开发，文件通过 export 暴露接口，通过 import 引用其他文件的内容。Vue.js 的源码结构优雅、层级严谨，而这一切都离不开 export 和 import。

举例来说，Vue.js 的入口文件是/src/index.js，该文件开头如下：

```
import Vue from './instance/vue'
import installGlobalAPI from './global-api'
import { inBrowser, devtools } from './util/index'
import config from './config'
```

index.js 从其他 4 个文件引用了必要的变量和方法。进入其引用的第一个文件/src/instance/vue.js 中，我们可以看到，其采用了 export default 命令：

```
export default Vue
```

也有不采用 export default 方式暴露接口的，比如在/src/filters/array-filters.js 中采用以下语法对外暴露了三个函数方法：

```
export function limitBy(…){},
export function limitBy(…){},
export function limitBy(…){}
```

在 src/filters/index.js 中引入上述三个函数：

```
import { orderBy, filterBy, limitBy } from './array-filters'
```

30.2.2　let

代码中任何一对花括号（{ 和 }）中的语句集都属于一个块，其中定义的所有变量在代码块外都是不可见的，我们称之为块级作用域。在 ES 5 及之前的版本中均不存在块级作用域，只有全局作用域和函数作用域。由此带来的问题很多，比如内层变量可能覆盖外层变量、用于计数的循环变量泄露为全局变量等。以往，开发者往往要通过闭包等方式来模拟块级作用域，尤为烦琐。let 关键词的出现为 JavaScript 带来了期盼已久的块级作用域。

1. let 的使用

let 的作用是声明变量，用法类似于 var。与 var 所不同的是，let 声明的变量只在 let 命令所在的代码块内有效。代码示例如下：

```
{
  let a = 'ddfe';
  var b = 'didiFamily';
}

a;  // ReferenceError: a is not defined
b;  // 'didiFamily'
```

在用 let 声明的 a 的代码块之外调用 a 会报错，而用 var 声明的 b 则可以返回正确值。这表明 let 声明的变量只在其所在代码块内有效。

let 很适合用来声明循环变量，代码示例如下：

```
for (let i = 0; i <  10; i++) {
```

```
  …// i 只在 for 循环体内有效
}
console.log(i);  // ReferenceError: i is not defined
```

let 不允许重复声明。在相同作用域内，重复声明同一个变量会报错。代码示例如下：

```
{
  let a = 'ddfe';
  let a = 'didiFamilyt' // 报错
}

{
  let a = 'ddfe';
  var a = 'didiFamilyt' // 报错
}
```

let 不存在变量提升。因此，变量需要在声明后才可以使用，否则会报错。代码示例如下：

```
console.log(foo); // 输出 undefined
console.log(bar); // 报错 ReferenceError

var foo = 'ddfe';
let bar = 'ddfe';
```

let 存在暂时性死区（temporal deadzone，以下简称 TDZ），即指在当前代码块内如果使用了 let 声明变量，则在该条声明语句之前，该变量不可用。代码示例如下：

```
if (true) {
  // TDZ 开始
  tmp = 'ddfe'; // ReferenceError
  console.log(tmp); // ReferenceError

  let tmp; // TDZ 结束
  console.log(tmp); // undefined

  tmp = 123;
  console.log(tmp); // 123
}
```

2. let 在 Vue.js 中的使用

由于使用 let 具有严谨、不易发生错误同时含有块级作用域等诸多优点，在 Vue.js 中运用 let

命令的情况比比皆是。比如：

```
// 在 src/config.js 文件中保存了全局配置信息
let delimiters = ['{{', '}}']
let unsafeDelimiters = ['{{{', '}}}']
```

再比如，在/src/batcher.js 文件中，runBatcherQueue 函数中的 for 循环变量也是采用 let 声明的：

```
function runBatcherQueue (queue) {
  // do not cache length because more watchers might be pushed
  // as we run existing watchers
  for (let i = 0; i < queue.length; i++) {
    …
  }
  queue.length = 0
}
```

30.2.3　const

1. const 的使用

const 用于声明一个常量，一旦声明，常量的值便不能再被更改，进入只读模式。代码示例如下：

```
const PI = 3.141;
PI // 3.141
PI = 3;
PI // 3.141， 重新赋值无效，PI 值不变
```

在严格模式下，对已使用 const 声明的变量重新赋值甚至会报错，提示“TypeError: 'xxx' is read-only”。

使用 const 声明后不得再更改的特性，也意味着我们在声明变量时就必须初始化，不能留到以后再赋值。代码示例如下：

```
const ddfe;
ddfe = 'wonderful'; // 重新赋值无效
ddfe // undefined
```

在严格模式下，使用 const 声明变量不赋值会报错，提示“SyntaxError: missing = in const declaration”。

const 的作用域与 let 相同，只在声明所在的块级作用域内有效。

const 与 let 一样，同样不允许重复声明，不存在变量提升以及 TDZ。

此外，还需要注意的是，const 在声明复合类型的变量时，只能保证变量名指向的地址不变，并不保证该地址的数据不变（因为复合类型的变量名指向数据地址而不指向数据）。因此，如果使用 const 声明一个对象，并不能保证对象不可更改。代码示例如下：

```
const ddfe = {};
ddfe.age = 4;
ddfe.age  // 4
ddfe = {};// TypeError: "foo" is read-only
```

由上例可以看到，使用 const 声明一个对象 ddfe 后，对象内的属性仍然可以修改。但不可改变 ddfe 指向的对象地址。

如果真的想达到对象本身不可变的效果，则应当使用 Object.freeze 方法，在此不展开介绍。

2. const 在 Vue.js 中的使用

在 Vue.js 中对于一些常用的变量都采用了 const 方式声明。比如在 src/transition/transition.js 中，对 transition 的动画类型的声明便采用了 const 命令。代码示例如下：

```
const TYPE_TRANSITION = 'transition'
const TYPE_ANIMATION = 'animation'
```

再比如，很多正则检测用的变量，也都使用 const 声明，因为这些都是一经声明就不再更改的常量。代码示例如下：

```
// src/compiler/compile.js
const bindRE = /^v-bind:|^:/
const onRE = /^v-on:|^@/
const dirAttrRE = /^v-([^:]+)(?:$|:(.*)$)/
const modifierRE = /\.[^\.]+/g
const transitionRE = /^(v-bind:|:)?transition$/


// src/filters/index.js
const digitsRE = /(\d{3})(?=\d)/g
```

30.3 object

1. Object.create

语法：Object.create(proto, [propertiesObject])

参数：

- proto —— 一个对象，作为新创建对象的原型。
- propertiesObject —— 可选。该参数对象是一个数组与值，该对象的属性名称将是新创建的对象的属性名称，值是属性描述符。

注：该参数对象不能是 undefined。另外，只有该对象自身拥有的可枚举的属性才有效，也就是说，该对象的原型链上属性是无效的。

抛出异常：如果 proto 参数不是 null 或对象值，则抛出一个 TypeError 异常。

用法：创建一个拥有指定原型和若干指定属性的对象。

使用举例：

```
// 下面的例子演示了如何使用 Object.create()来实现类式继承。这是一个单继承
// Shape - superclass
function Shape() {
  this.x = 0;
  this.y = 0;
}
Shape.prototype.move = function(x,y) {
  this.x += x;
  this.y += y;
  console.info("Shape moved.");
};

// Rectangle - subclass
function Rectangle() {
  Shape.call(this); // call super constructor
}
Rectangle.prototype = Object.create(Shape.prototype);

var rect = new Rectangle();

rect instanceof Rectangle // true
rect instanceof Shape // true

rect.move(1, 1); // Outputs, "Shape moved."
```

Vue.js 应用举例：

```
// src/global-api.js
// Vue.extend 用于创建基础 Vue 构造器的“子类”
Vue.extend = function (extendOptions) {
  ...
  var Sub = createClass(name || 'VueComponent')
  Sub.prototype = Object.create(Super.prototype)
  Sub.prototype.constructor = Sub
  ...
}
```

2. Object.keys

语法：Object.keys (obj)

参数：

- obj —— 返回该对象的所有可枚举自身属性的属性名。

用法：返回一个字符串数组，其元素来自于给定对象上可枚举的属性。这些属性的顺序与手动遍历该对象属性时的一致。

使用举例：

```
var arr = ["a", "b", "c"];
alert(Object.keys(arr)); // 弹出"0,1,2"
// 类数组对象
var obj = { 0 : "a", 1 : "b", 2 : "c"};
alert(Object.keys(obj)); // 弹出"0,1,2"
// getFoo 是一个不可枚举的属性
var my_obj = Object.create({}, { getFoo : { value : function () { return this.foo } } });
my_obj.foo = 1;

alert(Object.keys(my_obj)); // 只弹出 foo
```

Vue.js 应用举例：

```
// src/util/lang.js
// Mix properties into target object
export function extend (to, from) {
  var keys = Object.keys(from)
  var i = keys.length
  while (i--) {
    to[keys[i]] = from[keys[i]]
```

```
  }
  return to
}
```

3. Object.isExtensible

语法：Object.isExtensible(obj)

参数：

- Obj {Object}

用法：判断一个对象是否是可扩展的（是否可以在它上面添加新的属性）。在默认情况下，对象是可扩展的，即可以为其添加新的属性并且其__proto__属性可以被更改。Object.preventExtensions、Object.seal 或 Object.freeze 方法都可以标记一个对象为不可扩展的（non-extensible）。

注：在 ES 5 中，如果参数不是对象类型，将抛出一个 TypeError 异常。在 ES 6 中，非对象参数将被视为一个不可扩展的普通对象，因此会返回 false。

使用举例：

```
// 新对象默认是可扩展的
var empty = {};
Object.isExtensible(empty); // === true

// ...可以变得不可扩展
Object.preventExtensions(empty);
Object.isExtensible(empty); // === false

// 密封对象是不可扩展的
var sealed = Object.seal({});
Object.isExtensible(sealed); // === false

// 冻结对象也是不可扩展的
var frozen = Object.freeze({});
Object.isExtensible(frozen); // === false
```

Vue.js 应用举例：

```
// src/directives/public/for.js
```

```
// Cache a fragment using track-by or the object key
cacheFrag (value, frag, index, key) {
  var trackByKey = this.params.trackBy
  var cache = this.cache
  var primitive = !isObject(value)
  var id
  if (key || trackByKey || primitive) {
    id = getTrackByKey(index, key, value, trackByKey)
    if (!cache[id]) {
      cache[id] = frag
    } else if (trackByKey !== '$index') {
      process.env.NODE_ENV !== 'production' &&
      this.warnDuplicate(value)
    }
  } else {
    id = this.id
    if (hasOwn(value, id)) {
      if (value[id] === null) {
        value[id] = frag
      } else {
        process.env.NODE_ENV !== 'production' &&
        this.warnDuplicate(value)
      }
    } else if (Object.isExtensible(value)) {
      def(value, id, frag)
    } else if (process.env.NODE_ENV !== 'production') {
      warn(
        'Frozen v-for objects cannot be automatically tracked, make sure to ' +
        'provide a track-by key.'
      )
    }
  }
  frag.raw = value
}
```

4. Object.getOwnPropertyNames

语法：Object.getOwnPropertyNames(obj)

参数：

- Obj {Object}

用法：返回一个由指定对象的所有自身属性的属性名（包括不可枚举的属性）组成的数组。数组中枚举属性的顺序与通过 for...in loop（或 Object.keys）迭代该对象属性时的一致。数组中不可枚举属性的顺序未定义。

使用举例：

```
var arr = ["a", "b", "c"];
console.log(Object.getOwnPropertyNames(arr).sort()); // ["0", "1", "2", "length"]

// 类数组对象
var obj = { 0: "a", 1: "b", 2: "c"};
console.log(Object.getOwnPropertyNames(obj).sort()); // ["0", "1", "2"]

// 使用 Array.forEach 输出属性名和属性值
Object.getOwnPropertyNames(obj).forEach(function(val, idx, array) {
  console.log(val + " -> " + obj[val]);
});
// 输出
// 0 -> a
// 1 -> b
// 2 -> c

// 不可枚举的属性
var my_obj = Object.create({}, {
  getFoo: {
    value: function() { return this.foo; },
    enumerable: false
  }
});
my_obj.foo = 1;

console.log(Object.getOwnPropertyNames(my_obj).sort()); // ["foo", "getFoo"]
```

Vue.js 应用举例：

```
// src/observer/index.js
const arrayKeys = Object.getOwnPropertyNames(arrayMethods)
```

5. Object.defineProperty

语法：Object.defineProperty(obj, prop, descriptor)

参数：

- obj —— 需要定义属性的对象。
- prop —— 需被定义或修改的属性名。

- descriptor —— 需被定义或修改的属性的描述符。

用法：该方法直接在一个对象上定义一个新属性，或者修改一个已经存在的属性，并返回这个对象。

注：该方法的使用较为复杂，但其在 Vue.js 中有极为重要的应用——Vue.js 实现数据和视图联动的核心原理便在于该方法。在第 2 章中有 Object.defineProperty 方法的详细使用介绍，在此不再赘述。

Vue.js 应用举例：

```
// src/util/lang.js
export function def (obj, key, val, enumerable) {
  Object.defineProperty(obj, key, {
    value: val,
    enumerable: !!enumerable,
    writable: true,
    configurable: true
  })
}
```

30.4　函数柯里化

函数柯里化（curry）过程是通过逐步传参，在每一步中返回一个更具体的部分配置的函数的过程。通过不断传参的过程，我们可以实现对函数的高度复用。函数柯里化是利用闭包、高阶函数特性来实现动态创建函数、参数复用的过程。柯里化可以使得代码逻辑更加清晰、代码实现更加优雅。由于在 Vue.js 中大量应用了函数柯里化技术，本节我们就简要介绍一下柯里化的使用。

30.4.1　动态创建函数

通常，我们会使用以下方式来注册 DOM 事件：

```
var addEvent = function(el, type, fn, capture) {
  if (window.addEventListener) {
    el.addEventListener(type, function(e) {
      fn.call(el, e);
    }, capture);
```

```
  } else if (window.attachEvent) {
    el.attachEvent("on" + type, function(e) {
      fn.call(el, e);
    });
  }
}
```

使用这种方式的问题是每次调用 addEvent 方法时，都会执行一次判断来决定使用哪个方法注册事件。实际上，我们可以在页面载入后只进行一次判断，然后使用同一注册方法注册不同的事件。我们使用函数柯里化来实现：

```
var addEvent = (function(){
  if (window.addEventListener) {
    return function(el, sType, fn, capture) {
      el.addEventListener(sType, function(e) {
        fn.call(el, e);
      }, (capture));
    };
  } else if (window.attachEvent) {
    return function(el, sType, fn, capture) {
      el.attachEvent("on" + sType, function(e) {
        fn.call(el, e);
      });
    };
  }
})()
```

以上自执行代码首先会判断浏览器支持的事件注册方法，根据不同的注册方法，返回一个事件注册的函数赋值给 addEvent，以后调用 addEvent 方法注册事件时，内部就不会再次进行判断了，而是直接使用当前浏览器支持的事件注册方法来注册相应的事件。

30.4.2　参数复用

如果我们需要求 10 与任意数的和，则可能会这么写：

```
function sum(x, y) {
  return x + y
}
sum(10, 20); // 30
sum(10, 55); // 65
```